УДК 004.42
ББК 32.973
О 75

Автор-составитель О. И. Еськова, канд. техн. наук, доцент

Рецензенты: В. Д. Левчук, канд. техн. наук, доцент, зав. кафедрой автоматизированных систем обработки информации Гомельского государственного университета им. Ф. Скорины;
В. В. Бондарева, канд. техн. наук, доцент кафедры информационно-вычислительных систем Белорусского торгово-экономического университета потребительской кооперации

Рекомендовано к изданию научно-методическим советом учреждения образования «Белорусский торгово-экономический университет потребительской кооперации». Протокол № 2 от 13 декабря 2016 г.

О 75 **Основы** алгоритмизации и программирования : пособие для реализации содержания образовательных программ высшего образования I ступени. В 2 ч. Ч. 2 / авт.-сост. О. И. Еськова. – Гомель : учреждение образования «Белорусский торгово-экономический университет потребительской кооперации», 2018. – 104 с.
ISBN 978-985-540-439-3

Пособие предназначено для студентов специальности 1-28 01 01 «Экономика электронного бизнеса». Содержит теоретический материал и практические задания, решение которых поможет выработать умения и навыки самостоятельной разработки программ.

УДК 004.42
ББК 32.973

ISBN 978-985-540-439-3 (ч. 2)
ISBN 978-985-540-430-0

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

Вторая часть пособия по дисциплине «Основы алгоритмизации и программирования» охватывает шесть тем: «Указатели», «Функции», «Динамическая память», «Строки», «Структуры», «Файлы». Эти темы соответствуют второму семестру изучения дисциплины и углубляют знания об основах алгоритмизации и языке программирования С («Си»), которые студенты получили при изучении первой части пособия.

Тема 6 знакомит студентов с понятием указателя, специфичным для языка С. Хотя указатели отсутствуют во многих других языках программирования, овладение этим инструментом значительно способствует развитию представлений о работе с памятью и готовит студентов к восприятию понятия «ссылки», которое используется практически во всех современных объектно-ориентированных языках.

В рамках темы 7 затрагиваются многие важные вопросы: организация передачи управления в функцию и возрата из нее, способы передачи параметров, вводится понятие перегрузки функций. Кроме того, подробно обсуждается такой важный прием программирования, как рекурсия. Приводятся примеры рекурсивных алгоритмов.

Тема 8 посвящена обсуждению способов использования динамической памяти как в классическом языке С, так и в языке С++. Подробно разбираются примеры организации многомерных динамических массивов и использования многоуровневых указателей.

Тема 9 затрагивает различные аспекты работы со строками в языке С: ввод и вывод строк, обработка строк как массивов, а также специальные функции из библиотеки string.h. Эта тема частично повторяет материал тем «Массивы» и «Указатели», закрепляя изученный материал на примере строк.

Тема 10 посвящена созданию и использованию структур, объединяющих элементы различных типов данных. Рассматриваются массивы структур и алгоритмы их обработки. Большое внимание уделяется созданным на основе структур динамическим спискам (стек, очередь, дек), которые широко используются в реальной практике программирования.

Тема 11 знакомит студентов с принципами организации файлов и потоков, а также функциями открытия и закрытия файлов, чтения и записи. Успешное усвоение этой темы позволит им грамотно использовать файлы в разрабатываемых приложениях.

Все темы пособия включают наборы задач, часть которых предназначена для самостоятельной работы студентов.

ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ, ЗАДАЧИ ДЛЯ ПРАКТИЧЕСКИХ ЗАНЯТИЙ

ТЕМА 6. УКАЗАТЕЛИ

6.1. Понятие указателя

Указатель – это переменная, содержащая адрес в памяти. Указатель связан с определенным типом данных. Эта связь задается при объявлении указателя:

```
int *p;     // p – указатель на значение типа int
double *q;     // q - указатель на значение типа double
```

Для работы с указателями используются две операции:

& – операция взятия адреса переменной;
* – операция взятия значения по адресу (операция разыменования).

Пусть, например, имеется обычная переменная целого типа *x*, которая расположена по адресу 4560. Указателю на тип *int* можно присвоить адрес этой переменной:

```
int x=3;
p=&x;      //p получает адрес переменной x
```

В результате в ячейку *p* будет записан адрес 4560 (рисунок 6.1). Аналогично, если вещественная переменная *y* расположена по адресу 4568, то указателю *q* можно присвоить этот адрес:

```
double y=3.5;
q=&y;   //q получает адрес переменной y
```

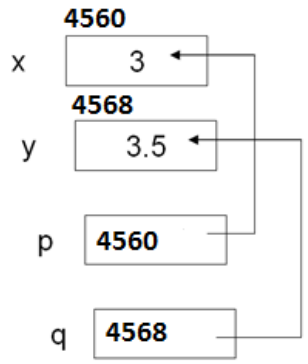


Рисунок 6.1 – **Иллюстрация понятия указателя**

При этом нельзя записать *p* = &*y*, так как это разные типы: *p* – указатель на целое, а &*y* – адрес вещественного числа.

Операция & (взятия адреса) не может применяться к константам, выражениям и регистровым переменным (с модификатором register).

Операция разыменования читается как «взять содержимое по адресу такому-то». То есть **p* – это содержимое по адресу, на который указывает *p*. И эту конструкцию можно использовать как обычную переменную целого типа:

```
*p=4; //в ячейку с адресом p записать 4
(*q)++; // ячейку с адресом q увеличить на 1
*q=*p/2.;      //содержимое по адресу p разделить на 2 и результат
               //записать по адресу q
```

При объявлении указателя конструкция со звездочкой имеет мнемонический смысл. То есть запись `int *p;` нужно читать как «содержимое по адресу *p* является целым числом».

Для ввода или вывода указателя с помощью функций scanf() и printf() используется формат %p.

При описании переменных-указателей рекомендуется включать в имена переменных префикс (суффикс) ptr (от pointer – указатель). Например, ptrOne, xPtr.

Существует три основных причины использовать указатели:

- передача в функцию адреса переменной, которая должна быть изменена;
- работа с массивом (эффективность);
- динамическое выделение и освобождение памяти.

В рамках данной темы разберем связь указателей и массивов, вопрос о передаче указателя в функцию рассмотрим в теме 7, а использование указателей для динамического выделения памяти – в теме 8.

6.2. Операции над указателями

Указателю на любой тип данных можно присвоить значение NULL (или 0). NULL – это «пустой указатель», который никуда не указывает:

```
p=NULL; //присваивание “пустого” указателя (или p=0;)
```

Указателю также можно присвоить значение другого указателя того же типа. В этом случае они указывают на одну и ту же переменную:

```
double z=0.5,*fPtr,*sPtr;
fPtr=&z;
sPtr=fPtr; //оба указателя содержат адрес z
```

Указатель можно увеличивать или уменьшать на целое число. При этом компилятор масштабирует приращение адреса в соответствии с типом, с которым связан данный указатель (т. е. для типа int приращение на 1 означает на самом деле увеличение адреса на 4 байта, а для типа double – на 8 байт). Таким образом, после увеличения на 1 указатель указывает на следующее число соответствующего типа:

```
p++; //увеличение указателя. Он теперь является адресом следующего
     //целого числа
q--; //q теперь является адресом предыдущего вещественного числа
p+=5; //увеличение адреса на 5 целых чисел вперед (на 20 байт)
```

При этом следует помнить, что унарные операции с одинаковым приоритетом выполняются справа налево.

То есть, например, ++*p означает, что содержимое по адресу *p* увеличивается на 1; *++p означает, что адрес *p* увеличивается на единицу (на самом деле на 4 байта – переходим к следующему целому числу в памяти), а затем берем содержимое, которое там находится.

Это иллюстрирует следующий код:

```
int *p,b;
b=50;
p=&b;
cout<<++*p;    // значение по адресу p увеличивается на 1 и выводится
               // Указатель не изменяется
cout<<*++p;    //выводится целое число, записанное по следующему адресу
               //(масштабированному на 4 байта). Указатель p изменится.
```

В результате на консоль будет выведено сначала число 51, а затем не определенное в программе содержимое следующей ячейки.

Можно найти разность двух указателей, если они указывают на один и тот же тип данных. Результатом будет количество чисел данного типа, которые можно разместить между этими указателями. Например:

```
int b=3, c=8;
int *bPtr=&b;
int *cPtr=&c;
cout<<"Адрес b= "<<bPtr<<"\n";
cout<<"Адрес c= "<<cPtr<<"\n";
cout<<"Разность указателей= "<<bPtr-cPtr<<"\n";
```

В этом фрагменте выводятся указатели (т. е. адреса двух целых чисел в памяти) и разность между ними, которая показывает, сколько чисел типа int находится между этими адресами. Пример результата выполнения этой программы показан на рисунке 6.2.

Рисунок 6.2 – **Результат вычисления разности указателей**

Понятно, что разность может быть и отрицательной (если первое число расположено в памяти после второго), но все равно учитывается масштабирование.

Нельзя получить разность указателей на разные типы данных (ошибка компиляции).

Указатели на один и тот же тип данных можно сравнивать:

```
if(bPtr>cPtr)
     cout<<"Адрес b больше!\n";
else
     cout<<"Адрес b меньше!\n";
```

Любой указатель можно сравнить со значением NULL (т. е. с указателем «в никуда»):

```
if(bPtr!=NULL)
     cout<<"Не пустой указатель!\n";
```

6.3. Указатели и массивы

Между указателями и массивами существует сильная взаимосвязь. Любую операцию, которую можно выполнить с помощью индексов массива, можно сделать и с помощью указателей. При этом вариант с указателями выполняется быстрее.

Более того, имя массива – это и есть указатель-константа на начало массива (на нулевой элемент). Объявим указатель и установим его на начало массива:

```
const int N=10;
int a[N],*ptr;
ptr=a; // эквивалентно ptr=&a[0];
```

После этого к *i*-му элементу массива можно обращаться любым из следующих способов:

```
a[i]    *(a+i)   *(ptr+i)    ptr[i]
```

Отличие состоит в том, что ptr – переменная-указатель, а имя массива – это константа-указатель. Поэтому изменять значение ptr можно, значение *a* – нельзя!

```
ptr++ ;    //можно
a++;       // нельзя!
```

Напомним, что при сложении указателя с константой выполняется масштабирование на `sizeof(тип_элемента)`, т. е. на количество байт, которые занимает один элемент массива.

Пользуясь указателем, можно, например, вывести элементы массива на консоль:

```
for(int i=0;i<N;i++)
     cout<<*(ptr+i)<<"\t";
cout<<"\n";
```

Однако этот вариант не даст выигрыша в быстродействии по сравнению с обращением a[i]. И в том, и в другом случае перед обращением к элементу массива вычисляется его адрес путем сложения адреса начала массива и масштабированного количества элементов: ptr + sizeof(int)*i (или a + sizeof(int)*i).

Если же сделать указатель подвижным, каждый раз увеличивая его на 1, то код будет работать быстрее, так как перед обращением к очередному элементу выполняется только сложение ptr + sizeof(int):

```
for(int i=0;i<N;i++,ptr++)
     cout<<*ptr<<"\t";
cout<<"\n";
```

Очевидно, что если два указателя указывают внутрь одного и того же массива, то разность этих указателей равна количеству элементов массива между ними.

Рассмотрим пример задачи на работу с массивом с помощью указателей.

Пример 6.1. В массиве из 10 целых чисел необходимо найти первый отрицательный и последний положительный элементы и поменять их местами:

```
//найти первый отрицательный и последний положительный элементы и
//поменять их местами
#include <iostream>
#include <time.h>
using namespace std;
void main()
{
     setlocale(LC_ALL,"rus");
     const int N=10;
     int a[N],*ptr;
     //ставим указатель на начало массива
     ptr=a; // эквивалентно ptr=&a[0]
     srand(time(0));
     for(int i=0;i<N;i++,ptr++)//в цикле увеличиваем указатель
          *ptr=rand()%21-10;
     cout<<"Исходный массив: \n";
     ptr=a; //указатель – на начало
     for(int i=0;i<N;i++,ptr++)
          cout<<*ptr<<"\t";
     cout<<"\n";
     int *fnegPtr,*lposPtr;
     //Поиск первого отрицательного
     fnegPtr=a; //указатель на начало массива
     int i=0;
     //пока массив не закончился и текущий элемет не отрицательный
     while(i<N&&*fnegPtr>=0)      {
          i++;//увеличиваем счетчик просмотренных
          fnegPtr++; //переставляем указатель на следующий
     }
     if(i==N) //проверка наличия отрицательных
     {
          cout<<"Нет отрицательных в массиве\n";
          system("pause");
          return;
     }
     lposPtr=a+N-1; //указатель на конец массива
     i=N-1; //счетчик тоже на конец
     //пока массив не закончился и текущий элемет не положительный
```

```
	while(i>=0&&*lposPtr<=0)
	{
		i--; //считаем количество оставшихся
		lposPtr--; //переставляем указатель на предыдущий
	}
	if(i<0) //проверка наличия положительных
	{
		cout<<"Нет положительных в массиве\n";
		system("pause");
		return;
	}
	//перестановка первого отрицательного и последнего положительного
	int tmp;
	tmp=*fnegPtr;
	*fnegPtr=*lposPtr;
	*lposPtr=tmp;
	//вывод преобразованного массива
	ptr=a;
	cout<<"Преобразованный массив: \n";
	for(int i=0;i<N;i++,ptr++)
		cout<<*ptr<<"\t";
	cout<<"\n";
	system("pause");
}
```

Что касается двумерных массивов, расположенных в стековой или в статической памяти, то при работе с ними можно учитывать тот факт, что элементы такого массива расположены в памяти подряд. Поэтому, если в алгоритме не нужно выделять строки (столбцы), то можно указатель поставить на начало массива и, увеличивая его на единицу, пройти весь массив. Например, нужно найти сумму элементов массива:

```
//сумма элементов двумерного массива
int sum=0;
int *aPtr=&a[0][0];
for(int i=0;i<ROW;i++)
	for(int j=0;j<COL;j++,aPtr++)
		sum+=*aPtr;
cout<<"Сумма элементов массива= "<<sum<<"\n";
```

Если же нужно работать со строками по отдельности, то следует учесть тот факт, что имя массива с одним первым индексом – это указатель на начало соответствующей строки:

a[i] – указатель на a[i][0].

Например, если нужно вывести элементы второй строки, то поставим указатель в начало этой строки, а затем будем двигаться, увеличивая этот указатель:

```
cout<<"Элементы второй строки:\n";
int *bPtr=a[1]; //эквивалентно bPtr=&a[1][0];
for(int j=0;j<COL;j++,bPtr++) //в цикле продвигаем указатель
	cout<<*bPtr<<"\t";
cout<<"\n";
```

Еще один пример – вывод первых элементов каждой строки. Указатель устанавливаем на первый элемент первой строки (номер 0), а затем увеличиваем на количество столбцов, попадая в следующую строку:

```
//первые элементы каждой строки
cout<<"Первые элементы строк:\n";
ptr=a[0];
for(int i=0;i<ROW;i++)
{
	cout<<*ptr<<"\t";
```

```
        ptr+=COL; //указатель величиваем на число столбцов:
                  //переход на новую строку
    }
    cout<<"\n";
```

Однако, если в двумерном массиве мы работаем со столбцами, то использование указателей никаких выигрышей в быстродействии не дает. Поэтому в последнем примере на самом деле лучше было обращаться к элементу массива через индексы: скорость выполнения программы та же, а код – намного понятнее.

6.4. Массивы указателей

Как и любой другой тип данных, указатель может быть элементом массива. Для объявления массива целочисленных указателей из десяти элементов следует написать:

```
int *x[10];
```

Далее присваиваем адрес целочисленной переменной var третьему элементу массива:

```
x[2] = &var;
```

Чтобы теперь получить значение по указателю, следует использовать конструкцию `*x[2]`.

Типичным примером применения массивов указателей является массив указателей на строки.

Строка – это массив символов, который заканчивается нуль-терминатором (символ «\0», код которого равен 0). Если записать в тексте программы строковый литерал (любой набор символов в двойных кавычках), то компилятор автоматически выделит память, в которую запишет коды этих символов, а затем добавит нуль-терминатор.

Если же объявить массив указателей на строки, то его можно инициализировать списком строковых литералов:

```
char *err[] = {"Cannot Open File\n", "Read Error\n", "Write Error\n","Media
Failure\n"};
```

После обработки компилятором этого кода, в памяти образуется конструкция, которая показана на рисунке 6.3. То есть строковые литералы размещаются в памяти, а элементам массива err присваиваются указатели на начало этих сток (на нулевой символ строки).

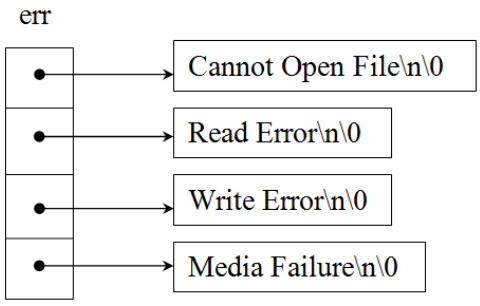


Рисунок 6.3 – **Массив указателей на строки**

Пусть теперь нужно написать программу, которая выводит сообщение об ошибке по ее номеру:

```
#include <iostream>
using namespace std;
void main()
{
```

```
    setlocale(LC_ALL,"rus");
    char *err[] = {"Cannot Open File\n", "Read Error\n", "Write Error\n",
"Media Failure\n"};
    int num;
    printf("Введите номер ошибки: ");
    scanf("%d",&num);
    if(num>=0&&num<4)
        printf ("Ошибка - %s", err[num]);
    //Аналогично: cout<<"Ошибка- "<<err[num];
    else
        printf("Неверный номер\n");
    system("pause");
}
```

Обратите внимание, что хотя err[num] – это указатель на начало строки, если выводить его по формату %s, то выводится вся строка (от этого указателя до нуль-терминатора) (если же выводить по формату %p – то выведется собственно адрес в памяти начала строки).

Более того, если использовать потоковый вывод `cout<<"Ошибка- "<<err[num];` то также выводится строка.

Массив указателей на строки также часто используют для сортировки строк, которая позволяет избежать их перестановки. То есть переставляются не сами громоздкие строки, а указатели на них. А отсортированный массив указателей потом используется для вывода строк в нужном порядке.

Пример 6.2. Пусть вводится массив из пяти строк. Нужно отсортировать его в порядке убывания первой буквы каждого слова.

Сами строки вводятся в двумерный массив символов *s* (количество столбцов такого массива равно максимально возможному числу символов в строке, включая символ «\0»). В массив указателей будем одновременно заносить адрес начала каждой строки. То есть после ввода данных мы получим в памяти ситуацию, аналогичную рисунку 6.4.

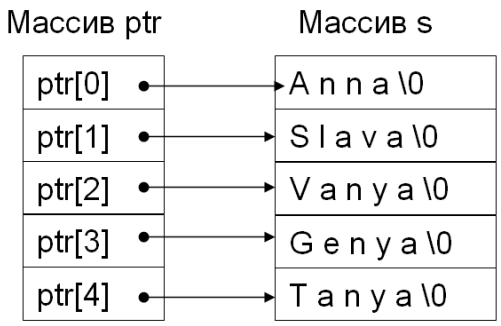


Рисунок 6.4 – **Массивы до сортировки**

В процессе сортировки будем переставлять не сами строки, а указатели. Необходимо получить результат, как на рисунке 6.5.

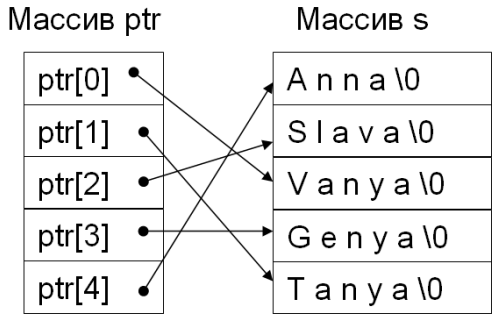


Рисунок 6.5 – **Массивы после сортировки**

При выводе массива используем указатели на начало каждой строки, т. е. выводим в упорядоченном виде:

```
#include <iostream>
using namespace std;
void main()
{
	setlocale(LC_ALL,"rus");
	const int SIZE=5;
	char s[SIZE][20], *ptr[SIZE], *tmp;
	cout<<"Введите "<<SIZE<<" строк:\n";
	for(int i=0;i<SIZE;i++)
	{
		cin>>s[i]; //ввод строки в массив строк
		ptr[i]=s[i]; //в массив указателей заносим адреса
			//первых символов строк
	}
//сортировка методом пузырька - переставляем указатели, а не строки
	for(int k=SIZE-1;k>0;k--)
		for(int i=0;i<k;i++)
			if(*ptr[i]<*ptr[i+1]) //сравнение первых букв слов
			{
				tmp=ptr[i];
				ptr[i]=ptr[i+1];
				ptr[i+1]=tmp;
			}
	//выводим строки в порядке указателей
	for(int i=0; i<SIZE;i++)
		cout<<ptr[i]<<"\n";
	system("pause");
}
```

6.5. Многоуровневые указатели

Поскольку указатель – это полноправная переменная, то можно создать указатель на указатель:

```
int **pPtr; //указатель на указатель
```

Такая переменная pPtr может содержать адрес ячейки, в которой находится указатель на переменную целого типа. Например, после выполнения следующего кода в памяти образуется картина, показанная на рисунке 6.6 (выводится на консоль число 5):

```
int a=5, *aPtr;
int **pPtr; //указатель на указатель
aPtr=&a;
pPtr=&aPtr;
cout<<"значение переменной a= "<<**pPtr<<"\n";
```

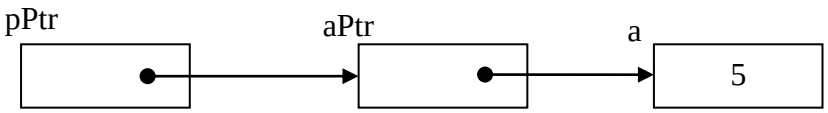


Рисунок 6.6 – **Указатель на указатель**

Так можно создавать указатели неограниченного уровня вложенности. При этом операция разыменования всегда интерпретируется справа налево.

Пример 6.3. Создается массив указателей на начало строк (строковые литералы содержат названия месяцев). Затем создается массив указателей на указатели, которые указывают на первый месяц сезона. Для продвижения по массиву указателей на указатели создается указатель на указатель на указатель. Получается конструкция, которая показана на рисунке 6.7.

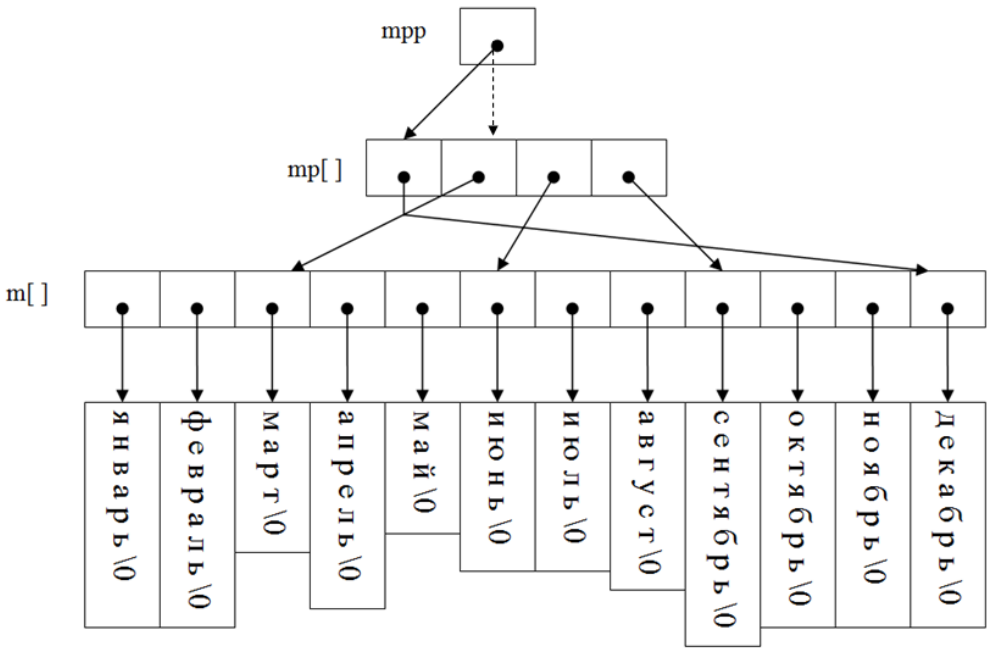


Рисунок 6.7 – **Многоуровневые указатели**

Код программы, который демонстрирует работу с этими данными, следующий:

```
#include <iostream>
using namespace std;
void main()
{
	setlocale(LC_ALL,"rus");
	//инициализация массива указателей
	char *m[]={"январь", "февраль", "март", "апрель", "май", "июнь", "июль",
"август", "сентябрь", "октябрь", "ноябрь", "декабрь"};
	//инициализация массива указателей на указатели массива m
	//(соответствующие первому месяцу сезона)
	char **mp[]={m+11,m+2,m+5,m+8};
	//вывод первого месяца сезона
	for (int i=0; i<4;i++)
		printf("%s\n",*mp[i]);
	//указатель для продвижения по массиву указателей
	char ***mpp;
	mpp=mp; //устанавливаем указатель mpp на начало массива mp
	// второй способ вывода начальных месяцев сезона
	for(int i=0; i<4; i++,mpp++)
		printf("%s\n",**mpp);
	system("pause");
}
```

Задачи

Задача 6.1. Создать массив из 10 целых чисел. Заполнить массив случайным образом в диапазоне от –45 до 45. Пользуясь указателями на массив целых чисел, посчитать процент положительных и отрицательных элементов массива.

Задача 6.2. Дан массив целых чисел (можно ввести значения элементов или задать случайным образом). Воспользовавшись указателями, поменяйте местами элементы массива с четными и нечетными индексами (т. е. те элементы массива, которые стоят на четных местах, поменяйте с элементами, которые стоят на нечетных местах).

Задача 6.3. Дан массив целых чисел (можно ввести значения элементов или задать случайным образом). Пользуясь двумя указателями, нужно поменять порядок элементов массива на обратный.

Задача 6.4. Объявить и заполнить случайными числами двумерный массив размерности 3 на 4. Пользуясь только указателями, распечатать элементы двумерного массива, а затем первые элементы каждой строки.

Задача 6.5. Объявить и заполнить случайными числами от –5 до 5 двумерный целочисленный массив размерности 5 на 6. Пользуясь только указателями, найти количество нулевых элементов в массиве.

Для самостоятельного решения

Задача 6.6. Даны два массива, упорядоченных по возрастанию: A[n] и B[m]. Сформируйте массив C[n+m], состоящий из элементов массивов A и B, упорядоченный по возрастанию.

Задача 6.7. Даны два массива: A[n] и B[m]. Необходимо создать третий массив, в котором нужно собрать:

- общие элементы двух массивов;
- элементы массива A, которые не включаются в B.

Задача 6.8. Создать массив указателей на строки, содержащие мужские и женские имена. Создать массив указателей на указатели, которые ссылаются на мужские имена в первом массиве. Аналогично создать массив указателей на указатели на женские имена. Пользуясь указателями, распечатать общий список имен, список мужских и список женских имен.

Задача 6.9. Дан отсортированный по возрастанию массив целых чисел, элементы которого не повторяются. Запросить у пользователя значение элемента. Если такой элемент имеется в массиве, то удалить его (сдвинуть влево оставшуюся часть массива). Если такого элемента нет, то вывести сообщение.

Задача 6.10. Массив целых чисел из 15 элементов заполнить случайными числами от –10 до 10. Найти среднее арифметическое элементов массива, которые расположены между первым отрицательным элементом и максимальным элементом в массиве.

Задача 6.11. Массив целых чисел из 10 элементов заполняется случайными значениями от 1 до 10. Программа должна удалить из массива все простые числа. Распечатать сначала исходный массив, а затем – преобразованный.

ТЕМА 7. ФУНКЦИИ

7.1. Понятие функции. Описание функции пользователя

Дальнейшим развитием идеи нисходящего проектирования программ является принцип *модульного программирования*. Суть его состоит в том, что программа конструируется из небольших частей, называемых модулями.

Преимущества модульного программирования:

- маленький фрагмент кода легче написать и отладить;
- модули можно использовать повторно в разных местах программы, это уменьшает объем кода и время, требуемое на разработку;

• модули можно объединить в библиотеки и предоставить другим программистам возможность их использования;

• в целом модульное проектирование повышает качество кода.

В языке С модули реализуются с помощью функций. *Функция* – именованный фрагмент кода, который может быть вызван многократно.

Мы уже использовали стандартные функции языка С (например, функции ввода и вывода printf() и scanf(), математические функции sqrt(), sin() и т. д.). Для использования стандартных функций нужно подключить соответствующий заголовочный файл библиотеки командой препроцессора #include.

Однако программист может написать и свою функцию, а затем вызывать ее из функции main (или из других функций) столько раз, сколько потребуется.

Описание функции состоит из заголовка и следующего за ним тела функции в фигурных скобках. В заголовке указывается тип данных результата, который возвращает функция, имя функции и в круглых скобках список формальных параметров функции (не обязательный). Таким образом, описание функции имеет следующий формат:

```
тип_результата имя_функции([список_параметров])
{    … //тело функции
     [return выражение;] //возврат результата
}
```

Тело функции – это собственно фрагмент кода, который выполняет некоторую определенную небольшую задачу. Внутри тела функции должен быть оператор return, который прекращает работу функции и возвращает некоторое значение в место ее вызова. Тип этого значения должен совпадать с типом результата функции (или неявно к нему преобразовываться). Оператор return может отсутствовать, если функция имеет тип результата void.

Пример 7.1. Функция возведения в квадрат может иметь следующий вид:

```
double sqr(double x)
{    return x*x;
}
```

Такое описание означает, что в функцию передается один параметр типа double (внутри функции он носит имя *x*). Сама функция называется sqr. По этому имени ее можно вызывать из других функций. А результат, который она возвращает, имеет тип double. Возврат значения происходит в тот момент, когда выполнение кода функции доходит до оператора return. Таким образом, если после оператора return находятся другие какие-то операторы, то они уже выполнены не будут.

Операторов return в теле функции может быть несколько, и тогда они фиксируют соответствующие точки выхода из модуля.

Пример 7.2. Данный пример демонстрирует, как работает функция с несколькими точками выхода. Функция signum возвращает 1, если аргумент положителен; –1 – если отрицателен; 0 – если равен 0:

```
int signum (int x)
{
     if (x>0) return 1;
     else if (x<0) return -1;
          else return 0;
}
```

Очевидно, что если в процессе выполнения этой функции окажется, что параметр положительный, то произойдет возврат по первому оператору return, а код, который записан в функции далее, игнорируется.

Если тип функции объявлен void, то функция никакого значения не возвращает. Внутри такой функции оператор return может быть без параметра или вообще отсутствовать. В этом

случае возврат в вызвавшую функцию происходит, когда процесс выполнения функции доходит до последней закрывающей скобки.

Пример 7.3. Функция, которая выводит на печать переданное ей значение:

```
void print(int x)
{
     cout<<"Результат= "<<x<<"\n";
}
```

Функция должна быть описана в программе до первого ее вызова. Например, если мы хотим использовать функцию возведения в квадрат в функции main, то сначала в файле программы размещаем описание функции sqr, а потом – функции main:

```
#include <iostream>
using namespace std;
//описание функции
double sqr(double x)
{
     return x*x;
}
//функция main() с вызовом функции sqr

void main()
{
     setlocale(LC_ALL,"rus");
     double z;
     z=sqr(3.2); //вызов функции
     cout<<"Результат= "<<z<<"\n";
     system("pause");
}
```

Если же удобнее поместить описание функции после вызова, то можно использовать *прототип функции* – заголовок функции, после которого ставится точка с запятой. При этом в прототипе могут указываться только типы формальных параметров, без имен. Прототип размещается в начале программы (перед первым вызовом функции), а само описание функции может тогда располагаться в любом месте программы.

Прототип сообщает компилятору интерфейс функции и дает возможность проверить правильность ее вызова (соответствие типа и количества параметров в вызове и в описании функции, соответствие типа результата).

Использование прототипа:

```
#include <iostream>
using namespace std;
//прототип
double sqr(double);
void main()
{
     setlocale(LC_ALL,"rus");
     double z;
     z=sqr(3.2); //вызов функции
     cout<<"Результат= "<<z<<"\n";
     system("pause");
}
//описание функции
double sqr(double x)
{
     return x*x;
}
```

Сформулируем некоторые правила описания функций:

• Функция должна выполнять только одну, небольшую задачу. Хорошим стилем программирования считается, если объем функции не превышает половины страницы кода.

• Имя функции должно иметь ясный смысл, отражать назначение функции. Например, sumOfArray() – функция для определения суммы всех элементов массива. Если не удается выразить суть функции одним словом, то, возможно, она выполняет более чем одну задачу. В этом случае проанализируйте, можно ли ее разбить на несколько функций.

• Если тип результата или параметров не указан, компилятор предполагает int.

Распространенной ошибкой программирования является указание в списке параметров double *x, y* вместо double *x*, double *y*. В первом случае параметр *y* по умолчанию считается int.

• Все функции в языке С являются равноправными. Нельзя описать одну функцию внутри другой.

7.2. Вызов функции. Передача параметров

Вызов функции имеет следующий формат:

```
имя_функции([список_аргументов]);
```

Эту конструкцию можно использовать как отдельный оператор. Тогда результат, возвращаемый функцией, игнорируется. Можно также каким-либо образом использовать результат функции. Например, присвоить его некоторой переменной или передать на консоль:

```
double s=sqr(2.5);
cout<<sqr(3.2)<<"\n";
```

Когда в процессе выполнения программы должен быть выполнен вызов функции, выполняются следующие действия:

• Создаются временные переменные для каждого формального параметра, который приведен в заголовке функции (под них выделяется место в памяти).

• Устанавливается соответствие между аргументами в вызове функции (фактическими параметрами) и формальными параметрами в ее заголовке.

Аргументы ставятся в соответствие параметрам по порядку в списке. Типы соответствующих аргументов и параметров должны совпадать.

В языке С аргументы в функцию передаются по значению, т. е. каждый параметр получает временную копию соответствующего аргумента.

• Управление передается в функцию.

• Код функции выполняется до тех пор, пока не встретится оператор return. Происходит возврат в вызвавшую функцию. При выходе из функции временные копии формальных параметров уничтожаются (память освобождается, они становятся недоступны).

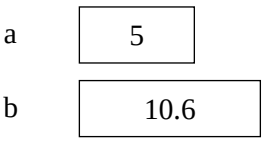
• Далее выполняется код, расположенный после вызова функции.

Приведем следующий пример описания и вызова функции:

```
#include <iostream>
using namespace std;
//описание функции
double fun (int a, double b)
{
     return a+b;
}
void main()
{
     setlocale(LC_ALL,"rus");
     int k=5;
     double d,l;
     //вызов 1
     d=fun(3,7.6);
```

Первый вызов функции:

Второй вызов функции:



```
	cout<<"d= "<<d<<"\n";
	//вызов 2
	l=fun(k,d);
	cout<<"l= "<<l<<"\n";
	system("pause");
}
```

В момент первого вызова создаются временные переменные *a* и *b*, для которых выделяется память. При этом *a* получает копию первого аргумента (3), *b* – второго (7.6). В качестве результата функции возвращается сумма этих значений (10.6), которая присваивается переменной *d*. Сами же переменные *a* и *b* после завершения первого вызова уничтожаются.

При втором вызове переменные *a* и *b* создаются вновь. При этом *a* получает копию переменной *k* (5), *b* – переменной *d* (10.6). В место второго вызова возвращается значение 15.6, которое присваивается переменной l. Параметры функции *a* и *b* уничтожаются.

Таким образом, фактические параметры *k* и *d* не могут быть изменены функцией, так как функция не имеет к ним доступа: она получает их временные копии.

Более того, даже если аргументы и параметры имеют одинаковые имена, это по сути разные переменные, имеющие разную область действия. Пусть, например, в нашей программе мы вместо имен *k* и *d* в функции main используем *a* и *b*:

```
#include <iostream>
using namespace std;
//описание функции
double fun (int a, double b)
{
	return a+b;
}
void main()
{
	setlocale(LC_ALL,"rus");
	int a=5;
	double b,l;
	//вызов 1
	b=fun(3,7.6);
	cout<<"b= "<<b<<"\n";
	//вызов 2
	l=fun(a,b);
	cout<<"l= "<<l<<"\n";
	system("pause");
}
```

Такое решение никак не повлияет на процесс выполнения программы. Результат будет тот же.

При вызове функции происходит автоматическое приведение аргументов к соответствующему типу по общим правилам преобразования типов в языке С.

Например, если вызвать функцию sqr, которая была описана ранее, с целым аргументом sqr(4), то целый аргумент автоматически преобразуется к типу формального параметра (double). То есть функция получит вещественное число 4.000.

Чтобы функция могла *изменить значение* некоторой переменной, следует передать ей *указатель* на эту переменную (т. е. фактически адрес переменной в памяти). Именно так мы поступали, когда использовали функцию scanf(): чтобы ввести данные в переменную *x*, мы в функцию передавали ее адрес: &*x*.

Пример 7.4. Функция, которая меняет местами значения двух переменных:

```
void swap(int *p,int *q)
{	int temp;
	temp=*p;
	*p=*q;
	*q=temp;
```

```
}
void main()
{    int x=4,y=9;
     swap(&x,&y); //вызов функции
}
```

7.3. Массивы и функции

Имя массива – это указатель на его нулевой элемент. Поэтому функция, получив имя массива, знает, где в памяти начинается массив, и может изменять все его значения. Массив как параметр функции можно описать двумя способами:

- указать имя и две пустые скобки : *a*[];
- объявить как указатель: **a*.

Кроме имени массива, в функцию следует передать его размерность.

Пример 7.5. Функция, которая заменяет в массиве все 0 на 1:

- 1-й вариант:

```
void convertArray(int mas[], int n) //замена 0 на 1 в массиве
{    for(int i=0;i<n;i++)
          if (mas[i]==0) mas[i]=1;
}
```

- 2-й вариант:

```
void convertArray(int *mas, int n) //замена 0 на 1 в массиве
{    for(int i=0;i<n;i++,mas++)
          if (*mas==0) *mas=1;
}
```

Пример 7.6. В программе используются три функции: инициализации массива случайными значениями; вывода массива на печать; замены всех нулей на единицы. Продемонстрированы разные способы передачи имени массива в функцию и последующего обращения к элементам массива (через индекс и через указатель). Следует понимать, что с точки зрения компилятора описание массива как int [] и int * полностью эквивалентны.

```
#include <iostream>
#include<time.h>
using namespace std;
//прототипы
void initArray(int[], int n);
void convertArray(int *, int n);
void putArray(int[], int n);
void main()
{
     setlocale(LC_ALL, "rus");
     const int N = 10;
     int a[N];
     srand(time(0));
     initArray(a, N);
     cout << "Исходный массив\n";
     putArray(a, N);
     convertArray(a, N);
     cout << "Результирующий массив\n";
     putArray(a, N);
     system("pause");
}
//описания функций
void initArray(int mas[], int n)
```

```
{
     for (int i = 0; i<n; i++, mas++)
          *mas = rand() % 11 - 5;
}
void putArray(int mas[], int n)
{
     for (int i = 0; i<n; i++)
          cout << mas[i] << "\t";
     cout << "\n";
}
void convertArray(int *mas, int n) //замена 0 на 1 в массиве
{
     for (int i = 0; i<n; i++)
          if (mas[i] == 0) mas[i] = 1;
}
```

Очевидно, что описанные в программе функции можно использовать для массивов любой размерности (а не только из 10 элементов). Например, вполне возможен такой вызов:

```
int x[]={1,2,0,0,3};
printArray(x, 5);
```

Работа с массивом внутри функции позволяет изменять сам объект, т. е. элементы массива. Если такое поведение нежелательно, то нужно объявить параметр-массив как константу:

```
void printArray(const int a[], int n);
```

В случае *двумерного массива* гибкость вызова несколько ограничена. Двумерный статический массив компилятор интерпретирует как массив из массивов-строк (рисунок 7.1).

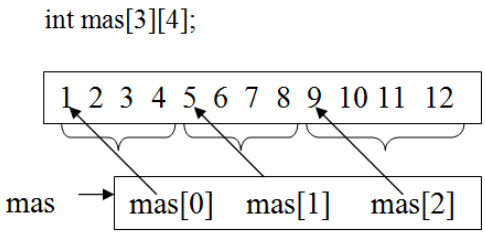


Рисунок 7.1 – **Интерпретация компилятором двумерного массива**

Имя двумерного массива из *N* строк и *M* столбцов – это указатель на указатель на начало массива из *M* элементов. Поэтому при передаче в функцию имени двумерного массива число столбцов будет фиксировано. Количество же строк можно передавать в качестве параметра.

Пример 7.7. Инициализировать двумерный массив размерности 3 на 4 случайными числами от –5 до 5, вывести его на консоль. Подсчитать сумму элементов массива.

1-й способ передачи двумерного массива в функцию – как двумерный массив с фиксированным числом столбцов:

```
//инициализация двумерного массива случайными числами
void initArray(int mas[][M],int n)
{
     for(int i=0;i<n;i++)
          for(int j=0;j<M;j++)
               mas[i][j]=rand()%11-5;
}
```

Вызов:

```
int a[N][M];
initArray(a,N);
```

Очевидно, что такую функцию можно использовать для массивов из *M* столбцов и любого числа строк.

2-й способ передачи двумерного массива в функцию – как указатель на массив-строку (опять же массив с фиксированным числом элементов):

```
//печать элементов двумерного массива в виде таблицы
void printArray(int (*mas)[M],int n)
{
     for(int i=0;i<n;i++)
     {
          for(int j=0;j<M;j++)
               cout<<mas[i][j]<<"\t";
          cout<<"\n";
     }
}
```

Вызов аналогичен:

```
printArray(a,N);
```

3-й способ передачи двумерного массива в функцию – передать указатель на нулевой элемент массива, количество строк и количество столбцов. При этом в алгоритме функции мы используем тот факт, что данный двумерный массив располагается в памяти подряд, строка за строкой. То есть фактически работаем с одномерным массивом.

```
//сумма элементов двумерного массива
int sumOfArray(int *mas, int n, int m)
{
     int sum=0;
     for(int i=0;i<n;i++)
          for(int j=0;j<m;j++)
          {
               sum+=*mas;
               mas++;
          }
     return sum;
}
```

Вызов:

```
sumOfArray(&a[0][0],N,M);
```

В этом случае число строк и число столбцов передаются в функцию как параметры. Поэтому эта функция будет работать для массивов любой размерности.

Представим программу, которая демонстрирует все три способа передачи двумерного массива в функцию:

```
#include <iostream>
#include <time.h>
#define N 3
#define M 4
using namespace std;
//прототипы
void initArray(int [][M],int);
void printArray(int (*)[M],int);
int sumOfArray(int *, int,int);
void main()
{
     setlocale(LC_ALL,"rus");
     int a[N][M];
```

```
	srand(time(0));
	initArray(a,N);
	printArray(a,N);
	cout<<"Сумма элементов массива="<<sumOfArray(&a[0][0],N,M)<<"\n";
	system("pause");
}
//описание функций
void initArray(int mas[][M],int n)
{
	for(int i=0;i<n;i++)
		for(int j=0;j<M;j++)
			mas[i][j]=rand()%11-5;
}
void printArray(int (*mas)[M],int n)
{
	for(int i=0;i<n;i++)
	{
		for(int j=0;j<M;j++)
			cout<<mas[i][j]<<"\t";
		cout<<"\n";
	}
}
int sumOfArray(int *mas, int n, int m)
{
	int sum=0;
	for(int i=0;i<n;i++)
		for(int j=0;j<m;j++)
		{
			sum+=*mas;
			mas++;
		}
	return sum;
}
```

В языке С массив не может быть результатом функции. Но зато можно вернуть указатель на первый элемент массива. Соответствующие примеры мы рассмотрим позже, когда будем говорить о динамическом выделении памяти.

7.4. Локальные и глобальные переменные. Область видимости

Переменные, описанные внутри функции, называются ***локальными***. Они создаются при входе в функцию и уничтожаются при выходе из нее. То есть область действия этих переменных ограничена пределами функции, в которой они объявлены. Очевидно, что параметры функции, которые передаются по значению, тоже являются локальными переменными.

Аналогично переменная, объявленная внутри блока, является локальной для этого блока. Например, если в цикле for объявляется переменная *i* типа int (for (int *i* = 0; *i* < *N*; *i*++)…), то после выхода из цикла она перестает существовать.

Одинаковые имена локальных переменных в разных функциях (блоках) не конфликтуют.

В функциях sqr и main используются локальные переменные *x* и *y*. Это абсолютно разные переменные, имеющие свою выделенную память и срок жизни:

```
#include <iostream>
using namespace std;
double sqr(double x)
{
	double y=x*x;
	return y;
}
```

```
void main()
{
     setlocale(LC_ALL,"rus");
     double x=2.5, y=4.5;
     cout<<"Результат 1= "<<sqr(x)<<"\n";
     cout<<"Результат 2= "<<sqr(y)<<"\n";
     system("pause");
}
```

Поскольку локальная переменная уничтожается при выходе из функции, передача ее адреса в качестве результата функции может в дальнейшем привести к ошибке.

Например:

```
int* f()
{
int a = 5;
return &a;     // нельзя, т.к. возвращается адрес в «никуда»!
}
```

Локальные переменные никак *не инициализируются* компилятором по умолчанию.

Переменная, объявленная вне любого блока, называется *глобальной*. Она имеет область действия от точки своего объявления до конца файла программы (рисунок 7.2). Глобальная переменная по умолчанию *инициализируется нулем* (или указателем NULL для указателей).

Любая функция может изменить значение глобальной переменной, и после выхода из функции эти изменения сохраняются.

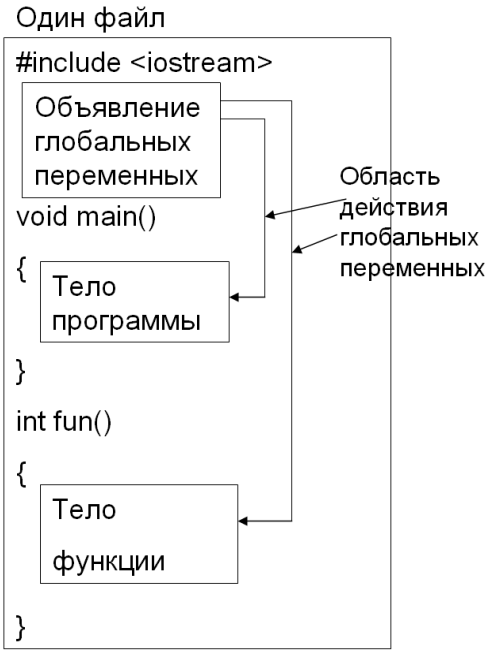


Рисунок 7.2 – **Область действия глобальной переменной**

Пример 7.8. Добавим в рассмотренную ранее программу глобальную переменную counter, в которой будем фиксировать количество вызовов функции sqr:

```
#include <iostream>
using namespace std;
double counter; //глобальная переменная инициализируется 0
double sqr(double x)
{
     double y=x*x;
     counter++;
     return y;
```

```
}
void main()
{
     setlocale(LC_ALL,"rus");
     double x=2.5, y=4.5;
     cout<<"Результат 1= "<<sqr(x)<<"\n";
     cout<<"Счетчик вызовов= "<<counter<<"\n";
     cout<<"Результат 2= "<<sqr(y)<<"\n";
     cout<<"Счетчик вызовов= "<<counter<<"\n";
     system("pause");
}
```

Очевидно, что к этой переменной имеют доступ все функции, определенные ниже точки ее описания.

Как правило, следует избегать использования в программах глобальных переменных. Поскольку любая функция может изменить значение такой переменной, сложно отследить все возможные изменения. Это дает потенциальную возможность возникновения ошибок.

Вместо использования глобальной переменной следует объявлять переменную внутри main и затем передавать ее (как параметр) в функции, которым она нужна.

Возможен *конфликт имен* локальной и глобальной переменной. Если локальная переменная имеет такое же имя, как и глобальная, то она «заслоняет» собой эту глобальную переменную. Глобальная переменная продолжает при этом существовать, но доступ к ней временно невозможен (до выхода из блока, в котором описана одноименная локальная переменная).

Если все же необходимо обратиться именно к глобальной переменной, то можно использовать оператор разрешения (::).

Пример 7.9. Использование оператора разрешения:

```
#include <iostream>
using namespace std;
int number=1001; //глобальная переменная
void shownumbers(int number) //параметр - локальная переменная
{
     cout<<"Локальная переменнная = "<<number<<"\n";
     cout<<"Глобальная переменная = "<<::number<<"\n";
}
void main()
{
     setlocale(LC_ALL,"rus");
     shownumbers(2002);
     system("pause");
}
```

В результате на консоль будет выведено:

```
Локальная переменная = 2002
Глобальная переменная = 1001
```

Следует различать понятия «область действия» и «область видимости» переменной. *Область действия* – это те фрагменты кода, где переменная существует (для локальных переменных: от точки объявления до конца блока, для глобальных – от точки объявления до конца файла). *Область видимости* – те фрагменты кода, где к переменной можно обратиться непосредственно, не используя оператор разрешения. В большинстве случаев область видимости совпадает с областью действия. Исключение составляет ситуация конфликта имен, как описано выше.

7.5. Статические переменные

В языке C++ можно перед переменной указать модификатор static. Для локальной переменной использование этого модификатора приводит к тому, что компилятор создает долго-

временную область для ее хранения почти таким же способом, как это делается для глобальной переменной.

Ключевое различие между статической локальной и глобальной переменными заключается в том, что статическая локальная переменная остается известной только в том блоке, в котором она была объявлена. Проще говоря, статическая локальная переменная – это локальная переменная, сохраняющая свое значение между вызовами функций. Этакая «постоянная память» для функции.

Использование статической локальной переменной гораздо предпочтительней, чем использование глобальной переменной, так как позволяет сузить область локализации возможной ошибки.

В качестве примера использования статических переменных приведем функцию, которая создает серию чисел, причем каждое следующее число основано на предыдущем (аналогично генератору псевдослучайных чисел):

```
#include <iostream>
using namespace std;
int series(void)
{
     static int series_num; //статическая переменная инициализируется 0
     series_num = series_num+23;
     return(series_num);
}
void main()
{
     setlocale(LC_ALL,"rus");
     cout<<"Серия чисел:\n";
     for(int i=0;i<10;i++)
          cout<<series()<<"\t";
     cout<<"\n";
     system("pause");
}
```

В данном примере переменная series_num существует между вызовами функций вместо того, чтобы каждый раз создаваться и уничтожаться, как обычная локальная переменная. Это означает, что каждый вызов series() может создать новый член серии, основываясь на последнем члене без глобального объявления переменной.

7.6. Ссылки

Ссылка – это понятие языка C++, в классическом языке C их не существовало.

Фактически *ссылка* – это псевдоним переменной. Она инициализируется при объявлении и изменению не подлежит.

Формат объявления:

```
тип &имя_ссылки = имя_переменной;
```

Тип ссылки и переменной должны быть одинаковыми.

Например:

```
int a = 5; //объявляем переменную
int &p = a; //объявляем ссылку. теперь p это псевдоним a
cout << a << ' ' << p << endl; //выведет 5 5
a = 6;
cout << a << ' ' << p << endl; //выведет 6 6
```

Чаще всего ссылка используется как параметр функции. При этом из функции можно напрямую работать с переменной, которую в нее передали.

На рисунке 7.3 проиллюстрирован принцип работы ссылки при передаче ее в функцию.

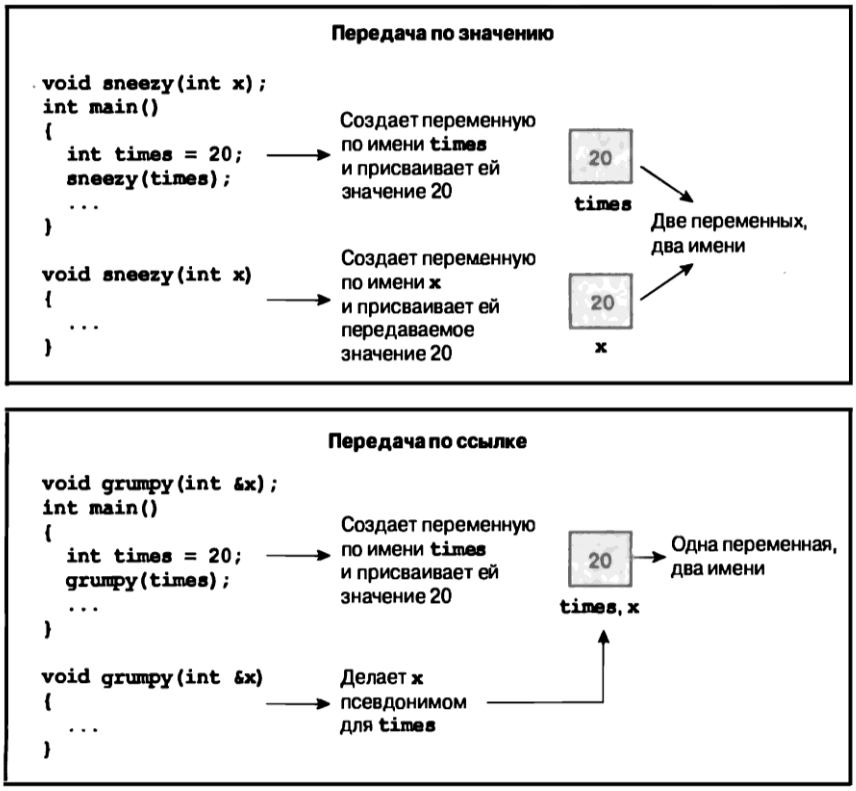


Рисунок 7.3 – **Принцип работы ссылки при передаче ее в функцию**

Пример 7.10. Рассмотрим функцию, которая меняет местами значения двух переменных. Сначала напишем вариант с использованием ссылок:

```
void myswap(int &a,int &b) //ссылки
{
     int temp=a;
     a=b;
     b=temp;
}

void main()
{
     int x=5,y=6;
     myswap(x,y); //передача в функцию переменных
     cout<<x<<' '<<y<<endl; //выведет 6 5
}
```

Для сравнения перепишем тот же пример с использованием указателей:

```
void myswap(int *a,int *b) //указатели
{
     int temp=*a;
     *a=*b;
     *b=temp;
}

void main()
{
     int x=5,y=6;
     myswap(&x,&y); //передача в функцию адресов переменных
     cout<<x<<' '<<y<<endl; //выведет 6 5
}
```

Преимущества ссылок перед указателями:

- код с использованием ссылок выглядит проще и лаконичнее;

• при передаче данных по ссылке не происходит копирование, что ускоряет вызов функции и экономит память.

Нужно учитывать тот факт, что при использовании ссылок любое изменение параметра внутри функции отражается на оригинале. Это снижает безопасность кода, повышает вероятность ошибки. Если необходимо защитить данные при передаче по ссылке и при этом выиграть в скорости, то можно использовать при описании функции модификатор const:

```
int sum_by_reference(const int &reference)
// функция, принимающая аргумент по ссылке
// const не дает изменить передаваемый аргумент внутри функции
```

Сформулируем *основные отличия ссылок от указателей*:

• Указатели ссылаются на участок в памяти, используя его адрес, а ссылки ссылаются на объект по его имени.

• Указатель – переменная, а ссылка – константа.

• При объявлении ссылка обязательно должна быть инициализирована, а указатель – не обязательно.

• Основное назначение указателей – это динамическое выделение памяти и эффективность при работе с массивами. Ссылки предназначены для организации прямого доступа к объекту (например, при передаче в функцию).

7.7. Стек вызовов

Стек – это структура данных, которая реализует принцип «последний пришел – первый вышел» (т. е. LIFO = Last In – First Out). Примером стека из реальной жизни может служить магазин автоматического оружия: патрон, который мы заталкиваем в магазин последним, выстрелит первым.

Существует другой способ наглядно представить себе стек. Вообразите пустой стакан, на дно которого мы кинули большую монету, а сверху – еще одну. Сможем ли мы теперь достать первую монету, не трогая второй? Очевидно, что нет. Нужно сначала извлечь монету, которую мы кинули последней, и тогда получим доступ к первой монете.

Стек имеет очень важное значение в программировании: именно в стеке сохраняется информация при вызове функций. При каждом вызове функции в стеке помещается фрагмент данных, который содержит:

• адрес возврата, т. е. адрес кода, куда нужно передать управление после завершения работы функции;

• значения параметров функции;

• значения локальных переменных функции.

В момент, когда работа функции закончена, из вершины стека извлекается этот фрагмент данных и содержащийся в нем адрес возврата используется для передачи управления в вызвавшую функцию.

На рисунке 7.4 проиллюстрирован наглядно процесс заполнения стека вызовов. Для простоты показан только адрес возврата для каждой функции (просто имейте в виду, что кроме адреса возврата, в стеке размещаются еще и локальные переменные – какой именно формат этих данных, нам пока не важно). Пусть функция main() в строке 6 вызывает функцию A(), которая в строке 2 вызывает функцию B(), а та – функцию C в строке 9. Поскольку адрес возврата – это адрес после вызова, то именно он и запоминается в стеке.

Когда выполнение функции доходит до конца (до последней закрывающей фигурной скобки или до оператора return), то адрес возврата извлекается из вершины стека вызовов (с удалением этого элемента). Таким образом, в каждый момент времени в вершине стека находится адрес возврата из текущей выполняемой функции.

В среде разработки Visual Studio можно открыть окно *Стек вызовов* (рисунок 7.5). Для этого необходимо задать команду *Отладка* → *Окна* → *Стек вызовов*.

В этом окне отображается состояние стека вызовов для текущего момента выполнения программы. Указывается имя каждой функции, которое может сопровождаться дополнительной информацией, такой как имя модуля, номер строки, смещение в байтах и др. (в зависимости от настроек).

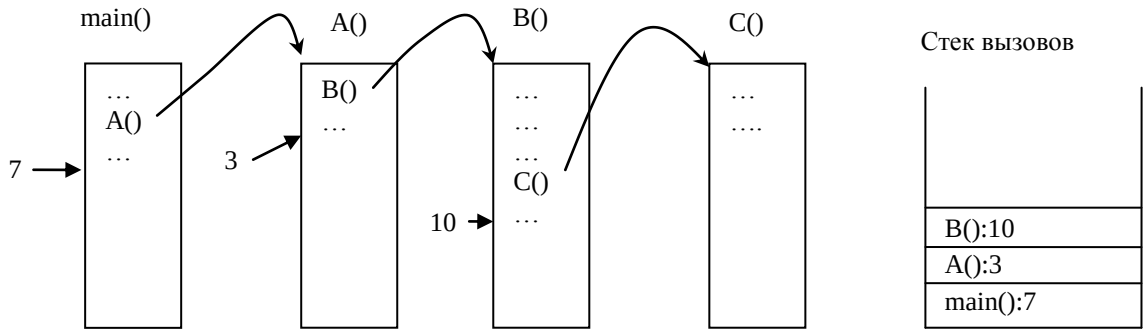


Рисунок 7.4 – **Состояние стека вызовов после трех последовательных вызовов функций**

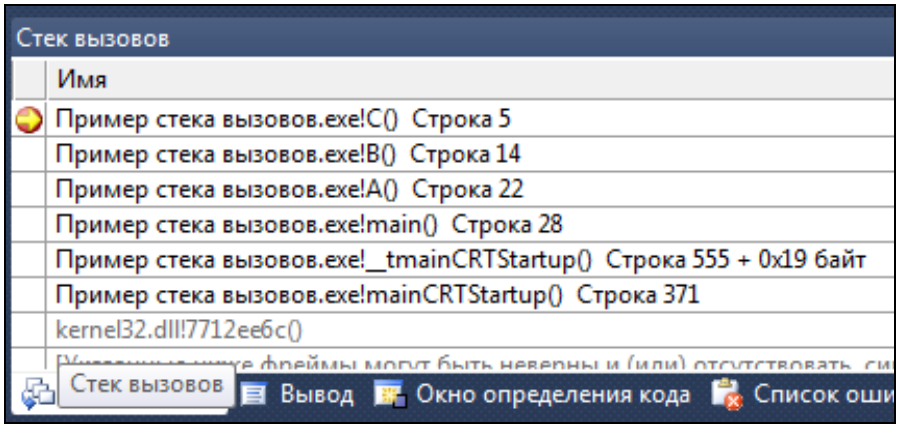


Рисунок 7.5 – **Окно стека вызовов в Visual Studio**

Использование окна стека вызовов особенно полезно в ситуации, когда функция может быть вызвана из разных мест. Используя это окно, очень просто определить, какая последовательность вызовов привела к текущему состоянию программы.

Кроме стека вызовов, в среде Visual Studio имеется окно *Иерархия вызовов*, которое позволяет узнать, где в исходном коде вызывается определенная функция.

Чтобы использовать окно *Иерархия вызовов*, выделите в исходном коде название функции, вызовите контекстное меню (правой кнопкой мыши) и выберите команду *Просмотреть иерархию вызовов*. В результате появляется окно, показанное на рисунке 7.6. В этом окне можно проследить логику вызовов данной функции и других функций, связанных с ней.

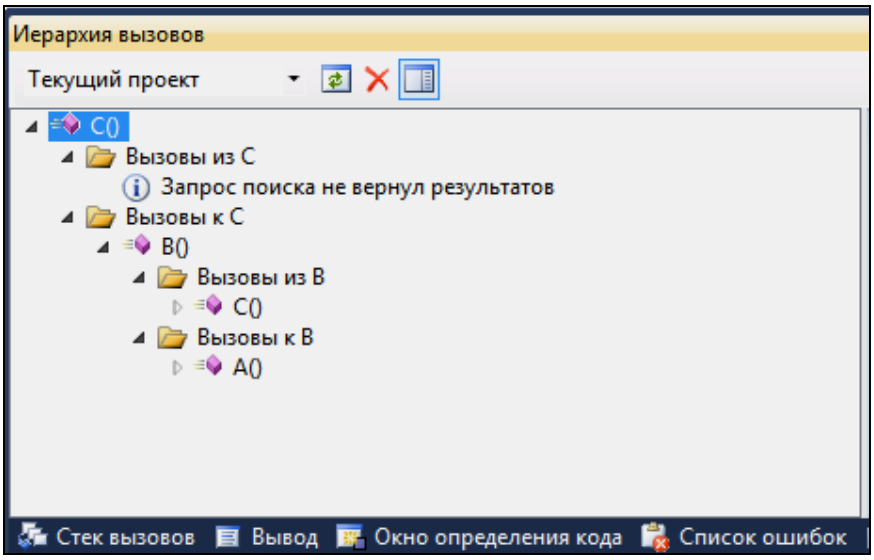


Рисунок 7.6 – **Окно *Иерархия вызовов***

7.8. Рекурсия

Рекурсия – это прием программирования, когда функция вызывает сама себя.

Пример 7.11. Рассмотрим вычисление факториала натурального числа *n*. *Факториалом* называется произведение всех целых чисел от 1 до *n*: $n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$.

При этом по определению 1!=1 и 0!=1.

Сначала напишем функцию вычисления факториала, которая использует цикл:

```
int factorial(int n)
{
     int f=1;
     for(int i=1;i<=n;i++) //перебираем числа от 1 до n
          f*=i; //и накапливаем их произведение
     return f;
}
```

А теперь дадим решение этой задачи, которое использует рекурсию. Для этого учтем, что можно записать соотношение: $n! = (n-1)! \cdot n$.

То есть факториал числа *n* можно легко вычислить через факториал предыдущего числа. Например, $5! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = (1 \cdot 2 \cdot 3 \cdot 4) \cdot 5 = 4! \cdot 5$.

Рекурсивная функция вычисления факториала:

```
int factorial(int n)
{
     if(n<=1) return 1; //выход из рекурсии
     return n*factorial(n-1); //вызов с меньшим значением параметра
}
```

Таким образом, каждый раз функция вызывает свою копию с меньшим значением параметра (решается задача все меньшей сложности). Так происходит до тех пор, пока параметр не станет равен 1. Тогда начинается цепочка возвратов в вызвавшие функции. На рисунке 7.7а показан процесс рекурсивных вызовов для расчета факториала 5, а на рисунке 7.7б – последовательность возвратов результата, которая в конечном итоге формирует искомое значение (120).

Основываясь на этом примере, сформулируем правила написания рекурсивных функций:

- Рекурсивная функция должна иметь терминальную ветвь, т. е. обычный возврат значения. Обычно эта терминальная ветвь располагается в начале функции, до рекурсивного вызова.
- Все рекурсивные вызовы должны вести к терминальной ветви (например, аргумент уменьшается при каждом вызове).

Проследить последовательность вычислений при рекурсии достаточно сложно. Это означает, что нужно научиться писать рекурсивные функции без этого представления, основываясь на логике сведения задачи к аналогичной, но более простой.

Любую рекурсию теоретически можно заменить на итерации (цикл). Рекурсия требует больше памяти и времени для выполнения вычислений, чем вариант задачи с использованием цикла. Это происходит потому, что каждый вызов функции требует выделения памяти под параметры и локальные переменные. Передача управления в функцию и возврат результата выполняются гораздо медленнее, чем вычисления в цикле.

Значит ли это, что рекурсия бесполезна? Разумеется, нет. Рекурсия может дать существенный выигрыш в красоте кода и понятности алгоритма для некоторых задач (вычисление факториала к ним не относится). Для таких задач рекурсивное решение заключается в нескольких строках, а вариант с использованием циклов был бы очень громоздким и трудным для восприятия.

Для того чтобы научиться писать рекурсивные алгоритмы, необходима практика. Рассмотрим несколько примеров.

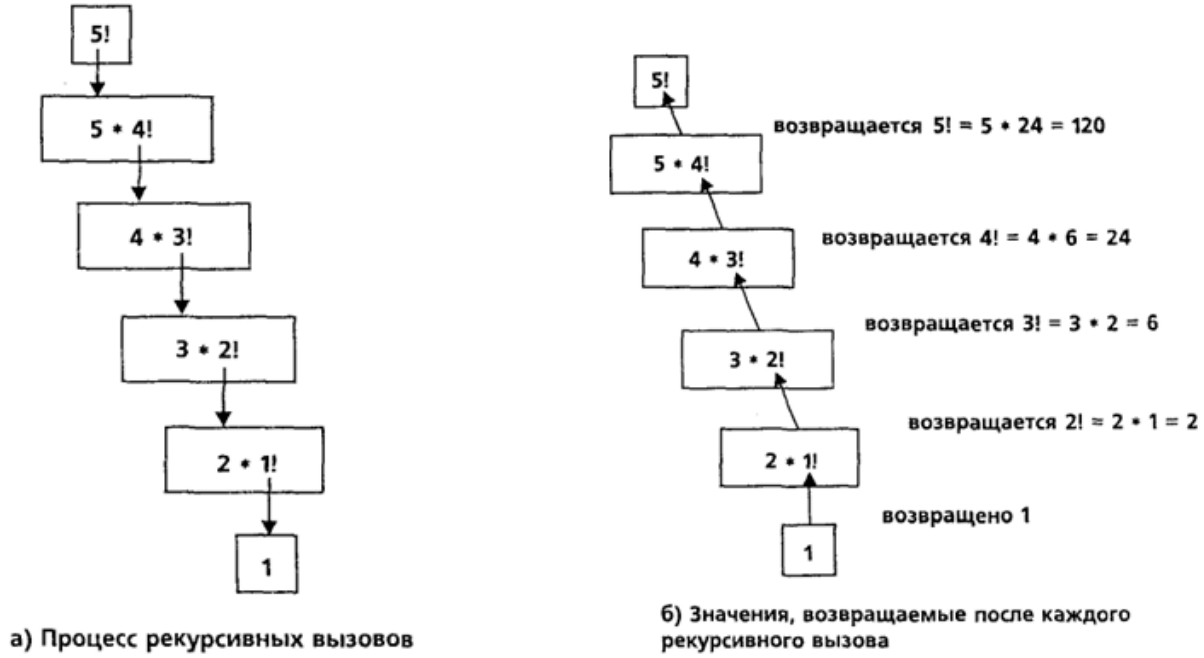


Рисунок 7.7 – **Последовательность вызовов и возвратов при расчете факториала 5**

Пример 7.12. *Числа Фибоначчи*. Числа Фибоначчи начинаются с 0 и 1. Каждое следующее число равно сумме двух предыдущих. То есть последовательность чисел Фибоначчи такова:

0, 1, 1, 2, 3, 5, 8, 13, 21, … .

Функция должна определять число Фибоначчи с номером *n*.

```
int fibonacci(int n)
{
     if(n==1) return 0;
     if(n==2) return 1;
     return fibonacci(n-2)+fibonacci(n-1);
}
```

Сначала идут терминальные ветки: для чисел с номерами 1 и 2 значения возвращаются непосредственно. Для остальных номеров выполняется два рекурсивных вызова, определяющих предыдущие значения, и возвращается их сумма.

С рекурсивными вызовами нужно обращаться осторожно, чтобы хватило ресурсов компьютера для реализации вызовов. В этом примере для расчета каждого числа Фибоначчи выполняется два вызова, поэтому количество вызовов растет как 2^n, и быстро выходит из-под контроля. Так, для 20-го числа Фибоначчи нужно выполнить 2^{20} вызовов (т. е. около миллиона), а для 30-го числа – 2^{30} вызовов (около миллиарда). Такие задачи называются задачами с экспоненциальной сложностью. Они могут привести к проблемам даже на современных быстродействующих компьютерах.

Пример 7.13. Дано натуральное число *n*. Следует вывести все целые числа от 1 до *n*. Использовать рекурсию.

```
void myprint(int i)
{
     if(i==0) return;
     myprint(i-1);
     cout<<i<<"\t";
}
```

Идея алгоритма проста: вызвать эту же функцию для вывода всех предыдущих чисел ($i - 1$), а потом вывести искомое число i. Терминальная ветка для числа 0: начинается цепочка возвратов.

Вызов этой функции: myprint(n) приводит к печати всех целых чисел от 1 до n.

Пример 7.14. Дано натуральное число. Вывести цифры числа справа налево (например, ввод: 179, вывод: 9 7 1; нельзя использовать массивы и циклы).

```
void printCif(int i)
{
     if(i==0) return;
     cout<<i%10<<"\t";
     printCif(i/10);
}
```

Выводим последнюю цифру числа (остаток от деления на 10). Затем отбрасываем эту цифру (делим нацело на 10) и вызываем эту же функцию для вывода цифр, находящихся вначале. Возврат начинается, когда деление на 10 даст 0.

Пример 7.15. *Бинарный поиск в отсортированном массиве*. Пусть для определенности массив упорядочен по неубыванию. Обозначим границы области поиска: сначала $lb = 0$ – нижняя граница и $ub = n - 1$ – верхняя граница. Выберем ее середину: $m = (lb + ub) / 2$. Если искомый элемент меньше, чем средний (elem < $a[m]$), то поиск осуществляется в левой части области (от середины), если больше – то в правой. Если средний элемент оказался равен искомому, то возвращаем его индекс.

Если же искомого элемента нет в массиве, то наступит момент, когда левая граница поиска станет больше правой. В этом случае функция возвращает значение –1, которое означает, что поиск не дал результатов.

```
int binarysearch(int a[],int elem,int lb, int ub)
{
     if(lb>ub) return -1; //ничего не найдено
     int m=(lb+ub)/2;
     if(elem==a[m]) return m;
     if(elem<a[m]) return binarysearch(a,elem,lb,m-1); //поиск в левой
                                                       //части массива
     else return binarysearch(a,elem, m+1,ub); //поиск в правой части
                                               // массива
}
```

Вызов для массива размерности n элементов: binarysearch (a,elem,0,$n - 1$);

Пример 7.16. *Алгоритм быстрой сортировки*. Наиболее популярный в настоящее время алгоритм быстрой сортировки имеет ярко выраженный рекурсивный характер. Суть этого алгоритма следующая:

- из массива выбирается опорный элемент (обычно средний);
- все элементы, которые меньше него, перемещаем в левую часть массива; все элементы, большие опорного – в правую часть массива;
- затем функция сортировки вызывается для каждой из этих частей.

Функция должна принимать указатель на массив и два индекса, обозначающие нижнюю и верхнюю границу сортируемой области. Таким образом, заголовок функции будет выглядеть следующим образом:

```
void quickSort (int a[],int left,int right) {…}
```

Сформулируем алгоритм разделения сортируемой области на две части более подробно:

- запоминаем значение среднего элемента: $p = a[(\text{left} + \text{right}) / 2]$;
- вводим два текущих индекса i и j, сначала они указывают на левую и правую границу (i = left; j = right);

• перемещаем индекс *i* вправо (увеличение), пока не найдем элемент, больший или равный опорному: while(*a*[*i*] < *p*) *i*++;

• перемещаем индекс *j* влево (уменьшение), пока не найдем элемент, меньший или равный опорному: while(*a*[*j*] > *p*) *j*--;

• меняем местами элементы с индексами *i* и *j*.

Процесс продолжается до тех пор, пока индексы *i* и *j* не «разойдутся», т. е. пока *i* < = *j*.

На рисунке 7.8 этот алгоритм проиллюстрирован на примере массива из 14 элементов.

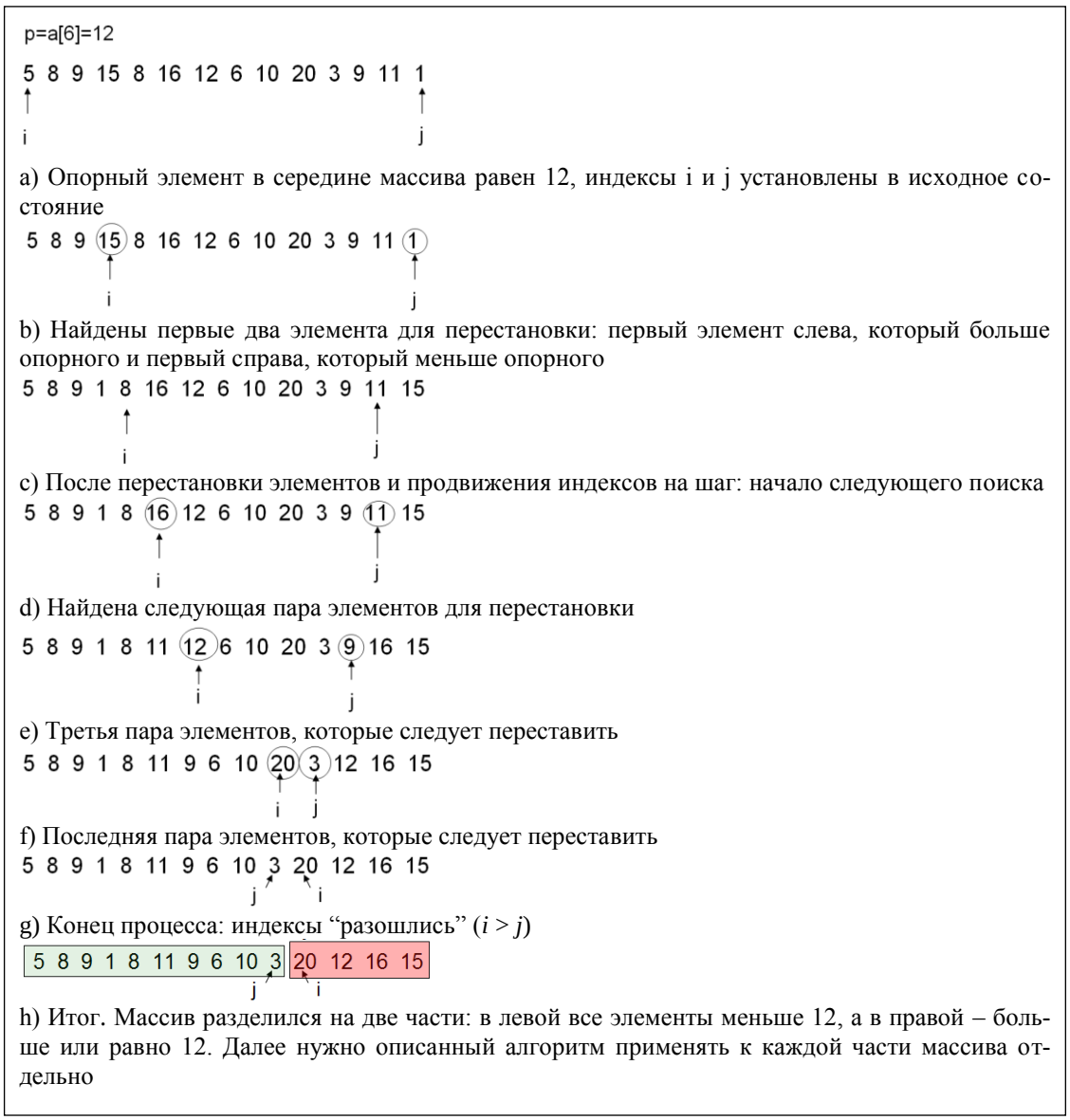


Рисунок 7.8 – **Иллюстрация разделения массива на две части**

Первый вызов этой функции должен учитывать исходные границы массива: `quickSort(a,0,n-1);`

Соответствующая функция быстрой сортировки:

```
void quickSort(int a[],int left,int right)
{
     int i=left,j=right;
     int temp, p=a[(left+right)/2];//опорный элемент
     //процедура разделения
     while(i<=j) //пока индексы не разошлись
     {
          while(a[i]<p) i++; //левый индекс двигаем до эл-та, >= опорн.
          while(a[j]>p) j--;//правый индекс двигаем до эл-та, <= опорн.
```

```
            if(i<=j) //если индексы еще не разошлись
            {
                  temp=a[i]; a[i]=a[j]; a[j]=temp;//меняем местами
                  i++; j--; //продвигаем индексы на шаг
            }
      }
      //рекурсивные вызовы
      if(j>left) quickSort(a,left,j);//отсортировать левый подмассив
      if(i<right) quickSort(a,i,right);//отсортировать правый подмассив
}
```

7.9. Перегрузка функций

Цель перегрузки функций состоит в том, чтобы создать несколько функций, которые обладают одним именем, но выполняются по-разному.

Пример 7.17. Пусть необходимо написать функцию, которая вычисляет площадь прямоугольника в квадратных сантиметрах. Назовем ее areaRectangle(). Мы хотим сделать так, чтобы эта функция была универсальной, поэтому создадим на самом деле две таких функции:

• Параметрами функции являются стороны прямоугольника в сантиметрах:

```
float areaRectangle (float a, float b)
{ return a*b;}
```

• Параметрами функции являются количество метров и сантиметров для каждой стороны (например, 2 м 43 см и 1 м 84 см):

```
float areaRectangle(float a_m, float a_cm, float b_m, float b_cm)
{return (a_m*100+a_cm)*(b_m*100+b_cm); }
```

Вызывается на исполнение в каждом конкретном случае та функция, список параметров которой соответствует фактическим параметрам:

```
void main()
{
      cout<<"S1= "<<areaRectangle(32,43)<<"\n"; //вызов варианта 1
      cout<<"S2= "<<areaRectangle(2,43,1,84)<<"\n"; //вызов варианта 2
}
```

Таким образом, перегруженные функции имеют одинаковые имена, но наборы их параметров должны различаться по количеству, типу или порядку элементов. Сочетание имени функции и последовательности типов ее параметров называется *сигнатурой функции*. Именно сигнатура является тем «лицом», по которому «узнает» функцию компилятор.

Например, все эти функции имеют разные сигнатуры (т. е. фактически, это разные функции, которые имеют одно имя):

```
void print(int a, int b);
void print(double a, double b);
void print(int a, double b, double c);
```

7.10. Шаблоны функций

Шаблоны функций в языке С позволяют создать общее определение функции, применяемой для различных типов данных.

В предыдущем пункте мы рассмотрели перегрузку функций, также позволяющую решить задачу создания функции с одним именем, которую можно применять к разным типам данных. Для этого нужно реализовать отдельную перегруженную версию функции для каждого типа.

Например, нужно создать функцию, которая вычисляет модуль числа, причем как для целых, так и для вещественных чисел:

```
int Abs(int n) //аргумент - целый
{return n < 0 ? -n : n; }
double Abs(double n) //аргумент - вещественный
{ return n < 0 ? -n : n; }
```

Используя шаблон, можно реализовать единственное описание, обрабатывающее значения любого типа:

```
template <typename myT>
myT Abs (myT n)
{ return n < 0 ? -n : n; }
```

Подробно разберем это описание. Ключевое слово template – начало конструкции шаблона. Ключевое слово typename означает «имя типа», а идентификатор myT (можно использовать любой идентификатор) является параметром типа. Конкретный тип myT определяется при вызове функции.

В описании также указано, что функция имеет один параметр типа myT (т. е. любого типа), а результат функции будет иметь тот же самый тип.

Допустим, программа вызывает функцию Abs и передает ей значения типа int:

```
cout << "Result - 5 = " << Abs(-5);
```

В момент вызова компилятор определяет тип фактического параметра. Поскольку это целое число, он считает, что myT эквивалентен типу int, и автоматически создает версию функции, где вместо myT подставляется int. Теперь функция будет выглядеть так:

```
int Abs (int n)
{ return n < 0 ? -n : n; }
```

Если же вызвать функцию с параметром типа double

```
cout << "Result 5.5 = " << Abs(5.5);
```

то будет создана такая версия функции:

```
double Abs (double n)
{ return n < 0 ? -n : n; }
```

Описание шаблона в тексте программы не вызывает генерацию кода само по себе. Код функции создается в тот момент, когда компилятор встречает вызов функции с параметром конкретного типа. Такой процесс носит название «создание экземпляра шаблона функции».

Следующий вызов с тем же типом данных параметра не вызовет генерацию новой функции, а использует ее уже существующую копию.

Если же в очередном вызове функции тип передаваемого параметра не совпадет ни с одним из предыдущих вызовов, то компилятор создаст новую версию функции.

В шаблоне может быть не один параметр типа, а несколько. Например, функция вычисления максимума двух чисел:

```
template <typename T, typename T1>
T myMax(T a, T1 b)
{
     return (a>b)?a:b;
}
```

Здесь два параметра, и они в принципе могут (но не обязаны) быть разных типов. Результат функции имеет тот же тип, что и первый параметр. Варианты вызовов:

```
int i=2,j=3,k;
double x=3.5, y=2.5,z;
k=myMax(i,j); //результат – тип int
k=myMax(i,x);  //результат  имеет  тип  int,  что  нежелательно,  т.к.  может
//произойти потеря данных
z=myMax(x,y); //результат – тип double
z=myMax(x,j); //результат – тип double
```

Следует обратить внимание на то, что каждый параметр типа, встречающийся внутри угловых скобок, должен обязательно появляться в списке параметров функции. В противном случае произойдет ошибка на этапе компиляции:

```
template <typename T1, typename T2>
T1 Max(T1 A , T1 B)
{ return A > B ? A : B; }
// ОШИБКА! список параметров должен включать T2 как параметр типа
```

Если в тексте программы есть описание обычной функции и шаблона с одним и тем же именем, то приоритет имеет обычная функция (переопределяет шаблон).

Например, пусть в тексте программы имеются следующие описания:

```
template <typename T>
T sum(T x, T y)
{ return x+y;}

void sum(char a,char b)
{ cout<<a <<" и "<<b; }
```

Тогда в вызове с параметрами, имеющими тип char, используется обычная функция sum. Для всех других типов параметров функция создается по шаблону:

```
sum('x','y'); //вызывается обычная функция
z=sum(5,6); //создается функция по шаблону
```

7.11. Указатель на функцию

Функция имеет физическое месторасположение в памяти, адрес начала которого иначе называется *точка входа*.

Имя функции без скобок и параметров является указателем-константой на функцию. Значением его служит адрес размещения функции в памяти (точка входа).

Можно провести аналогию с массивом: имя массива – адрес начала массива, который является константой.

Например, если имеется функция

```
int think(char);
```

то `think` – адрес этой функции в памяти.

Это значение адреса может быть присвоено некоторому указателю, и затем уже этот новый указатель можно применять для вызова функции.

Указатель на функцию определяется следующим образом:

```
тип_функции (*имя_указателя)(спецификация_параметров);
```

Следует знать, что тип функции и спецификация параметров (сигнатура) у объявляемого указателя и оригинальной функции должны совпадать.

Например, для функции think(), описанной выше, подойдет указатель:

```
int (*funPtr)(char);
```

Такому указателю можно присвоить адрес функции:

```
funPtr=think;
```

Примечание – Если приведенную синтаксическую конструкцию записать без первых круглых скобок, т. е. в виде `int *funPtr (char);`, то произойдет ошибка: компилятор воспримет ее как прототип некой функции с именем funPtr и параметром типа char, возвращающей значение указателя типа int *.

Инициализированный указатель на функцию можно применять для вызова функции:

```
k=(*funPtr)('*'); //эквивалентно вызову k=think('*');
k=funPtr('*'); //второй способ – тоже вызов k=think('*');
```

Пример 7.18. Пусть имеются две функции с одинаковыми сигнатурами и типами результата. Объявим аналогичный указатель на функцию, который можем инициализировать адресом одной или другой функции и затем использовать для вызова:

```
#include <iostream>
using namespace std;
int sum(int a, int b)
{    return a + b; }
int sub(int a, int b)
{    return a - b; }
void main()
{
     //объявление указателя на функцию
     int(*funPtr)(int, int);
     int a = 6, b = 5;
     funPtr = sum; //указатель получает значение адреса начала функции
                   //суммы
     //обращение к sum через указатель
     cout << a << " + " << b << " = " << (*funPtr)(a, b) << "\n";
     funPtr = sub; //указатель получает значение адреса начала функции
                   //разности
     //обращение к sub через указатель
     cout << a << " - " << b << " = " << funPtr(a, b) << "\n";
     system("pause");
}
```

Указатели на функции используются в основном для решения двух задач:

- передача адреса функции внутрь другой функции в качестве аргумента;
- создание массива указателей на функции для реализации меню.

В качестве примера решения первой из этих задач рассмотрим функцию расчета определенного интеграла, параметром которой является некоторая математическая функция.

Пример 7.19. *Расчет определенного интеграла*. Можно интерпретировать значение определенного интеграла $\int_a^b f(x)dx$ на интервале от a до b как площадь криволинейной трапеции под графиком функции $f(x)$ (на рисунке 7.9 это фигура, закрашенная серым цветом).

Для того чтобы подсчитать эту площадь, разобьем интервал $[a, b]$ на равные небольшие промежутки длиной h (чем меньше будет h, тем точнее рассчитано значение площади). Для левой границы такого интервала вычислим значение функции $f(x_i)$ – это будет высота прямоугольника. Таким образом, прямоугольник высотой $f(x_i)$ и шириной h левой верхней вершиной лежит на графике функции $f(x)$. Площадь прямоугольника подсчитать легко – это произведение его сторон: $f(x_i)*h$. А площадь криволинейной трапеции приблизительно равна сумме площадей прямоугольников:

$$\int_a^b f(x)dx \approx \sum_i f(x_i)\cdot h$$

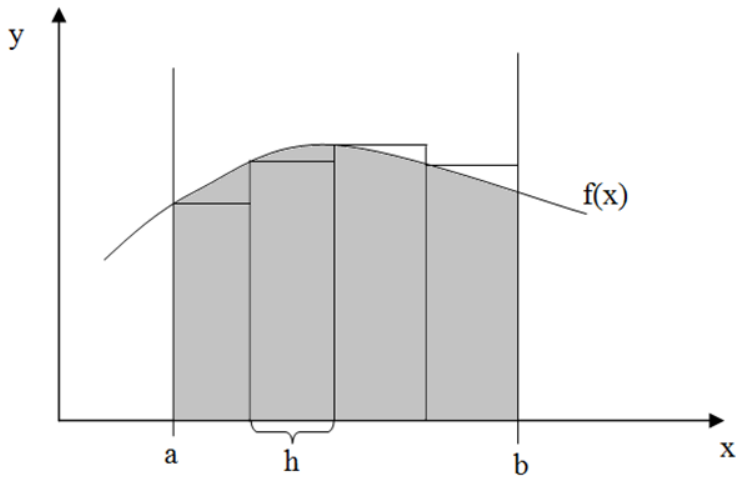


Рисунок 7.9 – **Графическая интерпретация расчета определенного интеграла**

Причем, чем меньше будет шаг разбиения *h* (или больше число интервалов разбиения), тем больше сумма площадей прямоугольников будет приближаться к истинному значению интеграла.

Следующая функция реализует расчет приблизительного значения интеграла этим методом:

```
//a и b – пределы интегрирования
//n – число интервалов разбиения
// pf – указатель на подынтегральную функцию
double integral(double a, double b, int n, double(*pf)(double))
{
     double s=0, h=(b-a)/n; //h- ширина интервала
     for(double x=a; x<b+h; x+=h)
          s+=pf(x)*h;
     return s;
}
```

Для того чтобы функция была универсальной и годилась для расчета интеграла различных математических функций, в качестве параметра функции использован указатель на функцию pf.

Этому указателю при вызове можно присвоить адрес любой функции, которая принимает аргумент типа double и возвращает результат типа double. Под это определение подходит большинство математических функций стандартной библиотеки (синус, косинус, экспонента, тангенс и пр.).

Например, вызов функции для расчета интеграла от синуса $\int_0^{\pi}\sin(x)dx$ будет выглядеть в функции main() так:

```
cout<<"Интеграл синуса от 0 до Пи= " <<integral(0.,3.1415,40,sin)<<"\n";
```

Пример 7.20. *Массив указателей на функции*. Рассмотрим теперь вторую из типичных задач, в которых используются указатели на функции: реализация меню с использованием массива указателей на функции.

Пусть есть ряд функций с одинаковым типом и набором параметров:

```
void vvod(void)
{
     cout<<"Ввод данных\n";
}
void udal(void)
{
```

```
     cout<<"Удаление данных\n";
}
void prosmotr(void)
{
     cout<<"Просмотр данных\n";
}
void quit(void)
{
     cout<<"Конец работы\n";
     system("pause");
     exit(0);
}
```

Объявим массив указателей на функции:

```
void(*funptr[4])(void)={vvod,udal,prosmotr,quit};
```

Массив имеет имя funptr и состоит из четырех элементов. Каждый элемент массива является указателем на функцию, которая не имеет параметров (тип void) и не возвращает результата. В качестве инициализирующих значений для элементов массива использованы точки входа наших однотипных функций.

Чтобы обратиться к *i*-му элементу массива, нужно написать вызов:

```
(*funptr[i])();
```

или

```
funptr[i]();
```

В процессе работы программы будем запрашивать выбор пользователя и вызывать соответствующую функцию из массива:

```
int answ;
do
{
     answ=menu(); //получение выбора пользователя
     funptr[answ-1](); //вызов соответствующей функции
}
while (true); //бесконечный цикл - выход по exit
```

Здесь используется еще одна вспомогательная функция – ввод выбора пользователя. Функция распечатывает меню и принимает от пользователя число от 1 до 4. Если же пользователь ошибся в выборе, то выводится соответствующее сообщение и опять повторяется ввод. Таким образом, функция menu() обеспечивает возврат корректного значения для обращения к массиву (возвращает число от 1 до 4):

```
int menu(void)
{
     setlocale(LC_ALL, "rus");
     int answer;
     do
     {
          cout << "1 - Ввод\n";
          cout << "2 - Удаление\n";
          cout << "3 - Просмотр\n";
          cout << "4 - Выход\n";
          cout << "Ваш выбор: ";
          cin >> answer;
          if (answer<1 || answer>4)
               cout << "Неверный выбор!\n";
```

```
    } while (answer<1 || answer>4);
    return answer;
}
```

Таким образом, использование массива указателей на функции позволяет избежать использования в программе оператора выбора или ветвления и упростить код.

Задачи

Задача 7.1. Написать функцию, которая возвращает истину, если ее аргумент – простое целое число, и ложь, если не простое. Простое число – это число, которое делится только на 1 и на себя (2, 3, 5, 7, 11 и т. д.).

Задача 7.2. Написать функцию frame, которая выводит на экран рамку. В качестве параметров функции должны передаваться координаты левого верхнего угла и размер рамки.

Задача 7.3. Напишите функцию, которая принимает целое число и возвращает количество цифр в числе и процент четных цифр.

Задача 7.4. Одномерный массив из 10 элементов инициализировать случайными числами от –9 до 9. Вывести исходный массив на консоль. Поменять местами первый отрицательный элемент и последний положительный. Преобразованный массив также вывести на консоль. При работе с массивом следует использовать только указатели. Оформить в виде отдельных функций: инициализацию массива; вывод на консоль; поиск первого отрицательного; последнего положительного; обмен местами двух элементов массива.

Задача 7.5. Описать функцию определения максимального элемента в одномерном массиве. С помощью этой функции найти максимум в каждой строке двумерного массива и вывести эти максимумы на консоль. Также с помощью этой функции найти максимальный элемент во всем двумерном массиве.

Для самостоятельного решения

Задача 7.6. Напишите функцию определения суммы элементов одномерного массива целых чисел.

Задача 7.7. Написать функцию, которая получает оценку студента по 100-балльной системе и возвращает эквивалент в 5-балльной системе. При этом если поступило число в диапазоне 90–100, то возвращает 5, для чисел в пределах 70–89 – возвращает 4, для диапазона 50–69 – результат 3. Если оценка менее 50, то возвращает 2. С помощью функции преобразовать из 100-балльной системы в 5-балльную оценки 8 студентов. Оценки вводить с клавиатуры или генерировать случайным образом.

Задача 7.8. Написать функцию, которая вычисляет объем и площадь поверхности параллелепипеда.

Задача 7.9. Написать функцию, которая в двумерном массиве вещественных чисел меняет знак элементов некоторого столбца на противоположный (номер столбца передавать в качестве параметра). Использовать функцию для преобразования матрицы размером 4×5. Выполнить два варианта такой функции: а) функция, предназначенная для массивов с числом столбцов, равным 5; б) универсальная функция, которую можно использовать при любом числе строк и столбцов.

Рекурсия

Задача 7.10. Функция получает в качестве параметров два целых числа A и B. Она выводит все числа от A до B в порядке возрастания, если A < B, или в порядке убывания в противном случае. Например, ввод: 5 1, вывод: 5 4 3 2 1.

Задача 7.11. Функция получает на вход целое число *n* и выдает true, если *n* является точной степенью двойки, и false – в противном случае. Операцию возведения в степень нельзя использовать.

Задача 7.12. Функция вычисляет сумму цифр целого числа, которое она получает в качестве параметра. Нельзя использовать массивы и циклы.

Задача 7.13. Вывести цифры числа слева направо. Например, ввод: 179, вывод: 1 7 9. Нельзя использовать массивы и циклы.

Для самостоятельного решения

Задача 7.14. Для вычисления наибольшего общего делителя (НОД) двух чисел можно использовать алгоритм Евклида:

```
                / НОД ( a-b , b ) если a > b
НОД (a,b) = <   a                 a = b
                \ НОД ( a , b-a )     a < b
```

Словами этот рекурсивный алгоритм можно выразить следующим образом: для того чтобы найти НОД двух чисел *a* и *b*, необходимо сравнить числа *a* и *b*. Если *a* = *b*, то НОД равен *a*. Если же одно из чисел больше другого, то нужно из большего вычесть меньшее, и повторно применить вышеизложенный алгоритм к разности и меньшему из чисел. Процесс завершается, когда числа становятся равными.

Реализовать алгоритм Евклида для определения наибольшего общего делителя двух целых чисел.

Задача 7.15. Функция получает на вход натуральное число *n*. Вывести все целые числа от 1 до *n*.

Задача 7.16. Дано слово, состоящее только из строчных латинских букв. Проверить, является ли это слово палиндромом. Вывести Yes или No. Например:

Ввод: radar Вывод: Yes
Ввод: yes Вывод: No

Задача 7.17. Написать рекурсивную функцию сортировки методом выбора.

Задача 7.18. Написать рекурсивную функцию, которая принимает двумерный массив целых чисел и количество сдвигов и выполняет круговой сдвиг массива вправо.

Например:

Дан массив:	Сдвигаем на три вправо:
4 5 6 7	5 6 7 4
1 2 4 1	2 4 1 1
4 5 6 9	5 6 9 4

Перегрузка и шаблоны

Задача 7.19. Напишите перегруженную функцию myArea, которая вычисляет площадь:

- квадрата (один параметр);
- прямоугольника (два параметра).

Задача 7.20. Написать функцию, которая меняет порядок элементов передаваемого массива на обратный. Внутри функции запрещено использовать вспомогательный массив. Использовать шаблон для реализации функций, работающих с различными типами массивов.

Задача 7.21. Написать перегруженную функцию mySum для реализации различных способов сложения дробных чисел:

- сложения десятичных дробей (с фиксированной точкой);
- сложения натуральных дробей.

Задача 7.22. Напишите функцию, которая принимает массив и количество элементов и возвращает среднее арифметическое всех элементов массива. Используйте шаблон, чтобы обеспечить работу как с целыми, так и с вещественными массивами. Аналогично напишите функцию для печати массивов разных типов.

Для самостоятельного решения

Задача 7.23. Напишите функцию «уплотнения» массива. Реализуйте возможность использования этой функции для различных типов массивов. Для целых и вещественных массивов эта функция удаляет все нули, сдвигая данные влево. Освободившиеся справа элементы заполняются значением 0. Для символьных массивов функция удаляет все пробелы, дополняя пробелами справа.

Задача 7.24. Реализовать 3 функции, каждая из которых принимает вещественное число и возвращает вещественное число. Первая функция вычисляет квадратный корень числа, вторая – куб числа, третья – синус числа. В главной функции создайте массив из 3 указателей на эти функции. Добавьте меню, в котором пользователь может выбрать желаемую операцию. Напишите программу без использования операторов if и switch.

Задача 7.25. Напишите 2 функции, каждая из которых принимает вещественный параметр и возвращает вещественное значение. Пусть первая функция вычисляет $y = x^2$, а вторая – $y = x \cdot 2 + 4$, где x – входной параметр, y – возвращаемое значение. Затем напишите функцию, которая принимает указатель на одну из этих функций, а также диапазон значений (от a до b) и выводит на экран 10 точек (координаты x и y) для этого диапазона. Напишите меню, которое иллюстрирует работу этих функций.

ТЕМА 8. ДИНАМИЧЕСКОЕ УПРАВЛЕНИЕ ПАМЯТЬЮ

8.1. Виды памяти

В процедурных языках программирования (таких, как С, Фортран, Паскаль, и т. д.) существуют три вида памяти: статическая, стековая и динамическая. Конечно, с физической точки зрения никаких различных видов памяти нет: оперативная память – это массив байтов, каждый байт имеет адрес, начиная с нуля. Когда говорится о видах памяти, имеются в виду способы организации работы с ней, включая выделение и освобождение памяти, а также методы доступа.

Статическая память

Статическая память выделяется еще до начала работы программы – на стадии компиляции и сборки. Статические переменные имеют фиксированный адрес, известный до запуска программы и не изменяющийся в процессе ее работы. Статические переменные создаются и инициализируются до входа в функцию main, с которой начинается выполнение программы.

Существует два типа статических переменных:

- глобальные переменные – это переменные, определенные вне функций (в описании которых отсутствует слово static);
- собственно статические переменные – это переменные, в описании которых присутствует слово static (они могут быть как локальными, так и глобальными).

Стековая, или локальная, память

Локальные, или стековые, переменные – это переменные, описанные внутри функции. Память для таких переменных выделяется в аппаратном стеке в момент входа в функцию или блок и освобождается в момент выхода из функции или блока.

Использование локальных переменных обеспечивает надежность работы программы, поскольку изменить такую переменную можно только внутри одной функции. Их можно использовать при рекурсивных вызовах функции.

Недостаток локальных переменных в том, что их нельзя использовать в качестве данных, разделяемых между несколькими функциями. К тому же размер аппаратного стека не бесконечен, стек может в один прекрасный момент переполниться (например, при глубокой рекурсии), что приведет к катастрофическому завершению программы. Поэтому локальные переменные не должны иметь большого размера. В частности, нельзя использовать большие массивы в качестве локальных переменных.

Динамическая память, или куча

Динамическая память (heap, куча) выделяется по запросу программы в процессе ее выполнения.

Под динамическую память отводится пространство виртуальной памяти процесса между статической памятью и стеком. Обычно стек располагается в старших адресах виртуальной памяти и растет в сторону уменьшения адресов. Программа и константные данные размещаются в младших адресах, далее располагаются статические переменные. Пространство выше статических переменных и ниже стека занимает динамическая память (рисунок 8.1).

адрес	содержимое памяти
0 4 …	код программы и данные, защищенные от изменения
...	статические переменные программы
…	динамическая память
max. адрес	стек ↑

Рисунок 8.1 – **Распределение оперативной памяти компьютера между видами памяти**

Структура динамической памяти автоматически поддерживается исполняющей системой языка С или С++. Динамическая память состоит из захваченных и свободных сегментов, каждому из которых предшествует описатель сегмента. При выполнении запроса на захват памяти исполняющая система производит поиск свободного сегмента достаточного размера и захватывает в нем отрезок требуемой длины. При освобождении сегмента памяти он помечается как свободный, при необходимости несколько подряд идущих свободных сегментов объединяются.

Если уже не используемый сегмент памяти так и остался помечен, как занятый, то может возникнуть «утечка памяти». Поэтому рекомендуется в конце программы всегда освобождать использованную память.

8.2. Функции языка С для работы с динамической памятью

В языке С для захвата и освобождения динамической памяти применяются стандартные функции malloc и free, описания их прототипов содержатся в стандартном заголовочном файле stdlib.h (в Visual Studio подключается автоматически). Имя malloc является сокращением от memory allocate – захват памяти. Прототипы этих функций выглядят следующим образом:

```
void *malloc(unsigned int n); //Захватить участок памяти размером n байт
void free(void *p); // Освободить участок памяти с адресом p
```

Функция malloc возвращает адрес захваченного участка памяти или NULL в случае неудачи (когда нет свободного участка достаточно большого размера). Функция free освобождает участок памяти с заданным адресом.

Для задания адреса используется указатель общего типа void*. После вызова функции malloc его необходимо привести к указателю на конкретный тип, используя операцию приведения типа. Например, в следующем примере захватывается участок динамической памяти размером в 4000 байтов, его адрес присваивается указателю на массив из 1000 целых чисел:

```
int *ptr;      // Указатель на массив целых чисел
. . .
ptr = (int *)malloc(1000 * sizeof(int));
```

Выражение в аргументе функции malloc равно 4000, поскольку размер целого числа `sizeof(int)` равен четырем байтам. Для преобразования указателя используется операция приведения типа (int *) от указателя обобщенного типа к указателю на целое число.

Функцию free() нужно использовать только с тем указателем, который был использован для ее выделения:

```
free(ptr);
```

Пример 8.1. У пользователя запрашивается размер массива, а затем выделяется память нужного размера. В эту память вводятся значения элементов массива, а затем введенный массив распечатывается (предполагается какая-то дальнейшая работа с этим массивом, например, сортировка):

```
#include <iostream>
using namespace std;
void main()
{
     setlocale(LC_ALL,"rus");
     int n, *ptr;
     cout<<"Введите размер массива: ";
     cin>>n;
     ptr=(int*)malloc(n*sizeof(int)); //динамич.выделение памяти из кучи
     int *tempPtr;
     tempPtr=ptr; //вспомогательный указатель для движения по массиву
     cout<<"Теперь введите элементы массива:\n";
     for(int i=0; i<n; i++,tempPtr++)
          cin>>*tempPtr;
     cout<<"Получен массив:\n";
     tempPtr=ptr; //опять вспомогательный указатель в начало
     for(int i=0;i<n; i++,tempPtr++)
          cout<<*tempPtr<<"\t";
     cout<<endl;
     free(ptr); //освобождение памяти
     system("pause");
}
```

8.3. Операции языка C++ для работы с динамической памятью

В языке C++ для захвата и освобождения динамической памяти используются операции new и delete. Они являются частью языка C++, в отличие от функций malloc и free, входящих в библиотеку стандартных функций языка C.

Пусть *T* – некоторый тип языка C или C++, *p* – указатель на объект типа *T*. Тогда для захвата памяти размером в один элемент типа *T* используется операция new:

```
T *p;
p = new T;
```

Например, для захвата восьми байтов под вещественное число типа double используется фрагмент кода:

```
double *p;
p = new double;
```

Если нужное количество памяти выделить не удалось, то результат операции – указатель NULL.

При использовании new, в отличие от malloc, не нужно приводить указатель от типа void* к нужному типу: операция new возвращает указатель на требуемый тип. Сравните два эквивалентных фрагмента на C и C++ (таблица 8.1).

Таблица 8.1 – **Сравнение кода выделения динамической памяти в языках C и C++**

Выделение памяти в языке C	Выделение памяти в языке C++
`double *p;` `p = (double*) malloc(sizeof(double));`	`double *p;` `p = new double;`

Конечно, второй фрагмент гораздо короче и нагляднее.

Операция new удобна еще и тем, что можно присвоить начальное значение объекту, созданному в динамической памяти (т. е. выполнить инициализацию объекта). Для этого начальное значение записывается в круглых скобках после имени типа, следующего за словом new. Например, в приведенной ниже строке захватывается память под вещественное число, которому присваивается начальное значение 1.5:

```
double *p = new double(1.5);
```

Этот фрагмент эквивалентен фрагменту

```
double *p = new double;
*p = 1.5;
```

С помощью операции new можно захватывать память под массив элементов заданного типа. Для этого в квадратных скобках указывается длина захватываемого массива, которая может представляться любым целочисленным выражением. Например, в следующем фрагменте в динамической памяти захватывается область для хранения вещественного массива размера *m* элементов:

```
double *a;
int m = 100;
a = new double[m];
```

Такую форму операции new иногда называют векторной.

Операция delete освобождает память, захваченную ранее с помощью операции new. Например:

```
double *p = new double(1.5); // Захват и инициализация
. . .
delete p; // Освобождение памяти
```

Если память под массив была захвачена с помощью векторной формы операции new, то для ее освобождения следует использовать векторную форму оператора delete, в которой после слова delete записываются пустые квадратные скобки:

```
int size=10;
double *a = new double[size]; // Захватываем массив
. . .
delete[] a; // Освобождаем массив
```

Приятная особенность оператора delete состоит в том, что при освобождении нулевого указателя ничего не происходит. Например, следующий фрагмент вполне корректен:

```
double *ptr = 0; // Нулевой указатель
bool b;
. . .
if (b) {
     ptr = new double[1000];
     . . .
}
. . .
delete[] ptr;
```

В указатель ptr вначале записывается нулевой адрес. Затем, если справедливо некоторое условие, захватывается память под массив. Таким образом, при выполнении операции delete указатель ptr содержит либо нулевое значение, либо адрес массива. В первом случае оператор delete ничего не делает, во втором освобождает память, занятую массивом. Такая технология применяется практически всеми программистами на языке C++: всегда инициализировать указатели на динамическую память нулевыми значениями и в результате не иметь никаких проблем при освобождении памяти.

Попытка освобождения нулевого указателя с помощью стандартной функции free может привести к аварийному завершению программы (это зависит от используемой C-библиотеки: нормальная работа не гарантируется стандартом ANSI).

Следует учитывать, что нельзя применять delete к одному и тому же указателю более одного раза.

Поэтому, если освобождение памяти может производиться в разных местах программы, после операции delete лучше присвоить указателю NULL:

```
delete[] ptr;
ptr=NULL;
```

Тогда, если далее будет попытка освобождения памяти, то это не вызовет ошибок.

Пример 8.2. Реализуем задачу предыдущего примера с помощью средств языка C++:

```
#include <iostream>
using namespace std;
void main()
{
     setlocale(LC_ALL,"rus");
     int n, *ptr;
     cout<<"Введите размер массива: ";
     cin>>n;
     ptr=new int[n]; //динамическое выделение памяти из кучи
     int *tempPtr;
     tempPtr=ptr; //вспомогательный указатель для движения по массиву
     cout<<"Теперь введите элементы массива:\n";
     for(int i=0; i<n; i++,tempPtr++)
          cin>>*tempPtr;
     cout<<"Получен массив:\n";
     tempPtr=ptr; //опять вспомогательный указатель в начало
     for(int i=0;i<n; i++,tempPtr++)
          cout<<*tempPtr<<"\t";
     cout<<endl;
     delete[] ptr; //освобождение памяти
     system("pause");
}
```

8.4. Многомерные динамические массивы

Использование массивов указателей на указатели позволяет создавать многомерные массивы, размещая их в динамической памяти.

Пример 8.3. *Двумерный динамический массив*. Организация двумерного динамического массива производится в два этапа. Сначала создается одномерный массив указателей, а затем каждому элементу этого массива присваивается адрес одномерного массива (строки):

```
int **pArr=new int *[m1]; //создание массива указателей на строки
for(int i=0; i<m1; i++)
     pArr[i]=new int[m2]; //размещение строки (указатель на нее – в массив)
```

Структура такого массива показана на рисунке 8.3. Следует отметить, что строки не обязательно располагаются в смежных участках памяти, как в случае статического массива.

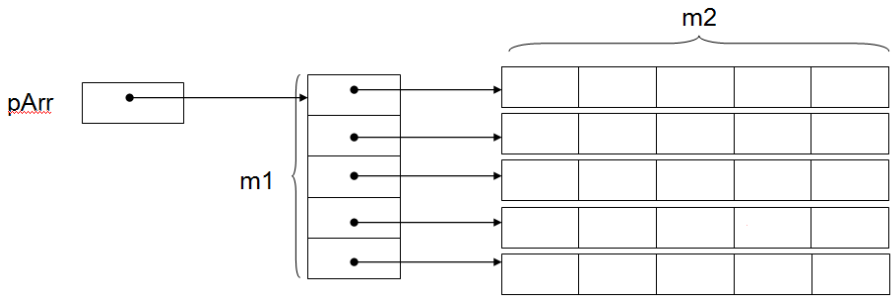


Рисунок 8.3 – **Организация двумерного динамического массива**

Обращение к элементу такого массива ничем не отличается от использования обычного двумерного массива. Например, записать в нулевую строку и первый столбец значение 100 можно так:

```
pArr[0][1]=100;
```

Освобождение памяти в конце работы программы должно выполняться в обратном порядке: сначала освобождается память, занимаемая строками (для доступа к ней используются адреса в массиве указателей на строке, затем удаляется сам массив указателей):

```
for(int i=0;i<m1;i++)
     delete [] pArr[i]; //удалениие i-й строки
delete [] pArr;         //удаление массива указателей на строки
```

Ниже приведен текст программы, в которой создается двумерный динамический массив. Сначала он весь заполняется нулевыми значениями. Затем первая строка заполняется натуральными числами от 1 до *m*2. Затем массив выводится на экран. В конце программы выполняется освобождение памяти:

```
#include<iostream>
using namespace std;
void main()
{
     setlocale(LC_ALL,"rus");
     int m1,m2;
     cout<<"Введите число элементов по первому измерению: ";
     cin>>m1;
     cout<<"Введите число элементов по второму измерению: ";
     cin>>m2;
     //создание массива
     int **pArr=new int *[m1];
```

```
        for(int i=0; i<m1; i++)
             pArr[i]=new int[m2];
        //заполнение нулями
        for(int i=0;i<m1;i++)
             for(int j=0;j<m2;j++)
                  pArr[i][j]=0;
        //Заполнение первой строки
        for(int j=0;j<m2;j++)
             pArr[0][j]=j+1;
        //печать массива
        cout<<"Получен массив:\n";
        for(int i=0;i<m1;i++)
        {
             for(int j=0;j<m2;j++)
                  cout<<pArr[i][j]<<"\t";
             cout<<"\n";
        }
        //удаление массива в порядке, обратном созданию
        for(int i=0;i<m1;i++)
             delete [] pArr[i];
        delete [] pArr;
        system("pause");
}
```

Пример окна консоли после выполнения данной программы показан на рисунке 8.4.

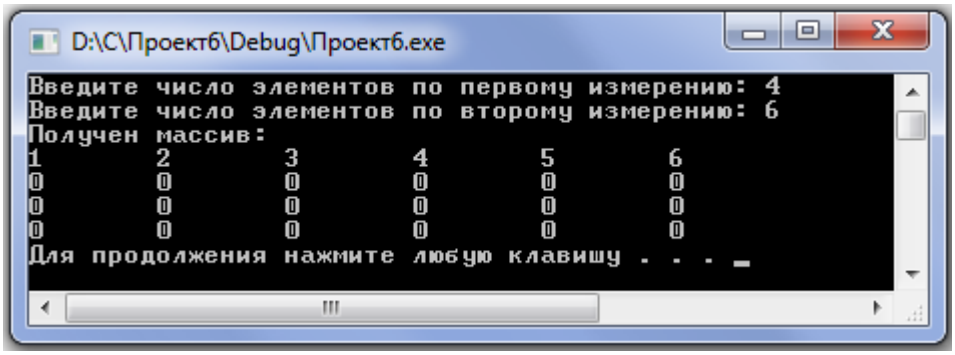


Рисунок 8.4 – **Результат создания двумерного динамического массива**

Пример 8.4. *Треугольный динамический массив*. Структура массива показана на рисунке 8.5. Строки массива имеют разную длину, начиная от значения *m*1, запрашиваемого у пользователя. Количество элементов в строке каждый раз уменьшается на 1.

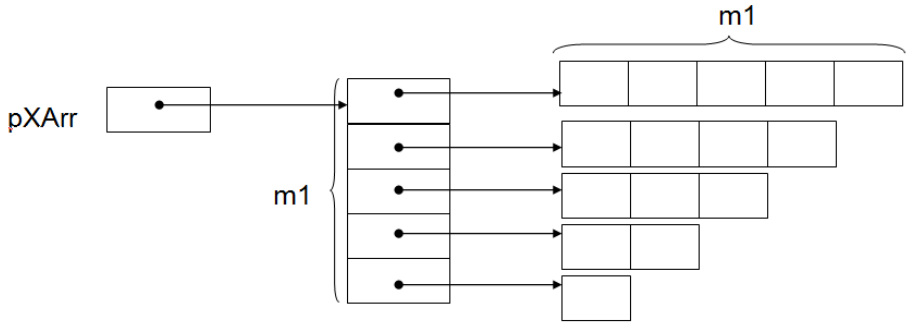


Рисунок 8.5 – **Треугольный динамический массив**

Создание массива аналогично предыдущему примеру: сначала создается массив указателей на строки, а затем в цикле память выделяется под каждую строку и инициализируется очередной элемент массива значением указателя на начало этой строки. Отличие лишь в том, что количество элементов в строке все время уменьшается на 1:

```
m2=m1;
int **pXArr=new int *[m1]; //создание массива указателей на строки
for(int i=0; i<m1; i++,m2--)
     pXArr[i]=new int[m2]; //создание строки
```

Обращение к элементу массива аналогично. Освобождение памяти выполняется в порядке, обратном его выделению:

```
for(int i=0;i<m1;i++)
     delete[] pXArr[i]; //освобождаем память, выделенную под строки
delete[] pXArr; //освобождаем память под массивом указателей
```

Полный текст программы, в которой создается треугольный массив, заполняется нулями и затем выводится на консоль, следующий:

```
#include<iostream>
using namespace std;
void main()
{
     setlocale(LC_ALL,"rus");
     int m1,wm,m2;
     cout<<"Введите число элементов по первому измерению: ";
     cin>>m1;
     m2=m1;
     //создание массива
     int **pXArr=new int *[m1];
     for(int i=0; i<m1; i++,m2--)
          pXArr[i]=new int[m2];
     //заполнение нулями и печать
     m2=m1;
     for(int i=0;i<m1;i++,m2--)
     {
          for(int j=0;j<m2;j++)
          {
               pXArr[i][j]=0;
               cout<<pXArr[i][j]<<"\t";
          }
          cout<<"\n";
     }
     //удаление массива в порядке, обратном созданию
     for(int i=0;i<m1;i++)
          delete [] pXArr[i];
     delete [] pXArr;
     system("pause");
}
```

Результат выполнения программы показан на рисунке 8.6.

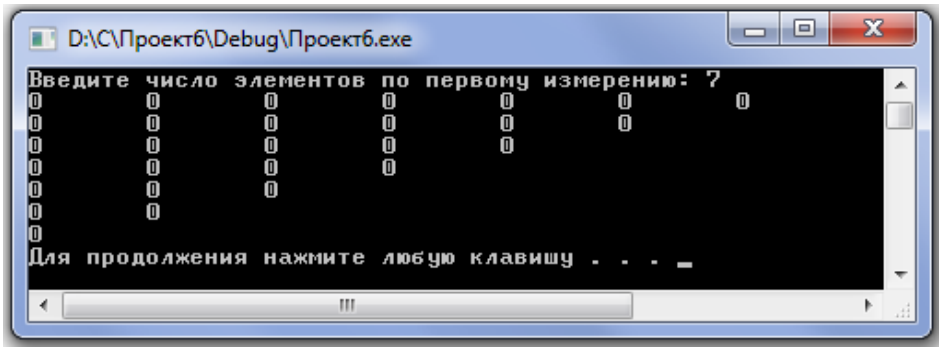


Рисунок 8.6 – **Результат создания треугольного динамического массива**

Пример 8.5. *Трехмерный массив*. При создании трехмерного массива сначала создается массив указателей на указатели. Индекс в этом массиве – это первый индекс трехмерного массива. Затем выделяется память под массивы указателей на целые числа (второй индекс из трех). И, наконец, выделяется память под массивы собственно целых чисел. Индекс в таком массиве – третий по счету. Массивы инициализируются указателями так, как показано на рисунке 8.7.

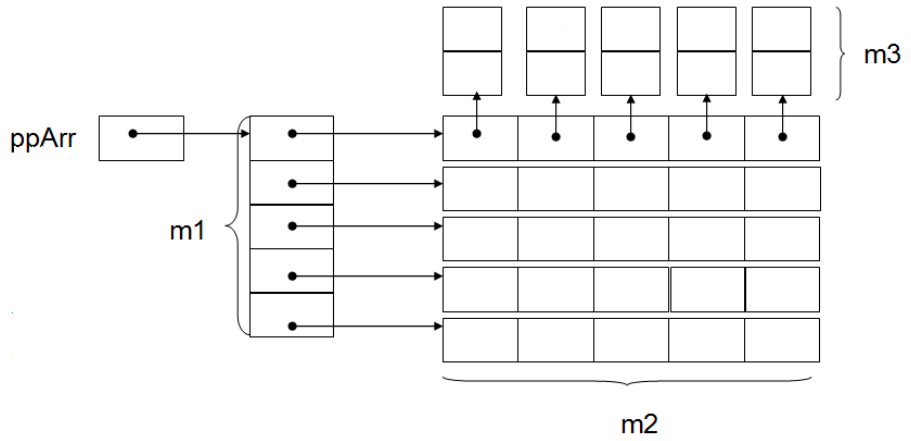


Рисунок 8.7 – **Трехмерный динамический массив**

Таким образом, для обращения к трехмерному массиву требуется указатель «в третьем поколении» ppArr:

```
int ***ppArr=new int **[m1]; //массив указателей на указатели
for(int i=0; i<m1; i++)
     ppArr[i]=new int *[m2]; //массив указателей
for(int i=0;i<m1;i++)
     for(int j=0;j<m2;j++)
          ppArr[i][j]=new int[m3]; //массив чисел
```

Для обращения к каждому элементу такого массива используются три индекса:

```
ppArr[0][0][0]=100;
```

Освобождение памяти также происходит в порядке, обратном созданию.

Далее приведен текст программы, создающей трехмерный массив и заполняющей его нулями. Далее нулевому по всем измерениям элементу присваивается значение 100, и выполняется вывод массива на консоль «по слоям»:

```
#include<iostream>
using namespace std;
void main()
{
     setlocale(LC_ALL,"rus");
     int m1,m2,m3;
     cout<<"Введите число элементов по первому измерению: ";
     cin>>m1;
     cout<<"Введите число элементов по второму измерению: ";
     cin>>m2;
     cout<<"Введите число элементов по третьему измерению: ";
     cin>>m3;
     //создание массива
     int ***ppArr=new int **[m1]; //первое измерение
     for(int i=0; i<m1; i++)
          ppArr[i]=new int *[m2]; //второе измерение
     for(int i=0;i<m1;i++)
          for(int j=0;j<m2;j++)
```

```
                ppArr[i][j]=new int[m3]; //третье измерение
        //заполнение нулями
        for(int i=0;i<m1;i++)
                for(int j=0;j<m2;j++)
                        for(int k=0;k<m3;k++)
                                ppArr[i][j][k]=0;
        //обращение к элементу
        ppArr[0][0][0]=100;
        //печать массива
        cout<<"Получен массив:\n";
        for(int i=0;i<m1;i++)
        {
                cout<<"Слой № "<<i<<"\n";
                for(int j=0;j<m2;j++)
                {
                        for(int k=0;k<m3;k++)
                                cout<<ppArr[i][j][k]<<"\t";
                        cout<<"\n";
                }
        }
        //удаление массива в порядке, обратном созданию
        for(int i=0;i<m1;i++)
                for(int j=0;j<m2;j++)
                        delete[] ppArr[i][j];
        for(int i=0;i<m1;i++)
                delete[] ppArr[i];
        delete[] ppArr;
        system("pause");
}
```

Результа выполнения программы представлен на рисунке 8.8.

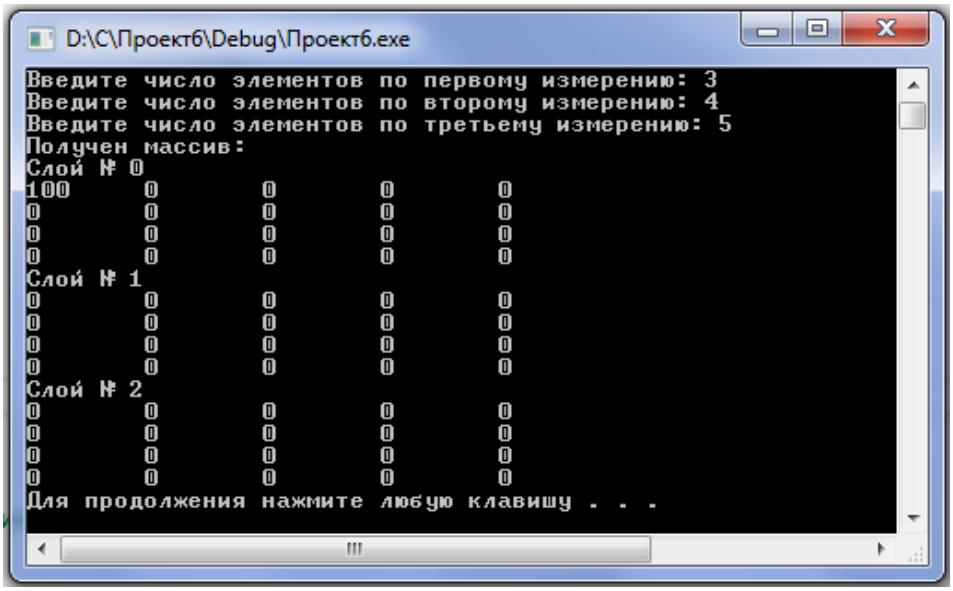


Рисунок 8.8 – **Результат создания трехмерного динамического массива**

Задачи

Задача 8.1. Сформировать и распечатать динамический одномерный массив (число элементов запросить у пользователя). Заполнить массив случайными числами от 0 до 20. Подсчитать сумму элементов массива.

Задача 8.2. Сформировать и распечатать динамический одномерный массив (число элементов запросить у пользователя). Заполнить случайными числами от –5 до 5. Удалить из массива все нулевые элементы. Новый массив должен занимать ровно столько памяти, сколько ему необходимо. Распечатать новый массив.

Задача 8.3. Запросить у пользователя размеры двумерного массива. Выделить память под массив, заполнив его случайными числами от –5 до 5. Найти максимальный элемент в каждой строке двумерного массива и вывести его на консоль.

Задача 8.4. Сформировать одномерный динамический массив и реализовать меню:

- добавление элемента в массив с заданной позиции;
- удаление заданного элемента массива;
- добавление нескольких элементов из массива, начиная с заданной позиции;
- удаление нескольких элементов из массива, начиная с заданной позиции.

Для самостоятельного решения

Задача 8.5. Создать треугольный динамический массив, в котором количество столбцов увеличивается на 1 от 1 до *m*, где *m* – количество строк. Заполнить массив случайными числами и вывести на экран.

Например:

m = 4:

```
10
34      17
67      3       8
11      89      26      1
```

Задача 8.6. Написать программу, которая проверяет, является ли введенная с клавиатуры квадратная матрица «магическим» квадратом. «Магическим» квадратом называется матрица, у которой сумма чисел в каждом горизонтальном ряду, в каждом вертикальном и по каждой из диагоналей одна и та же (рисунок 8.9). Размер матрицы вводит пользователь.

2	9	4
7	5	3
6	1	8

Рисунок 8.9 – **«Магический» квадрат**

Задача 8.7. Предположим, что характеристикой строки целочисленной квадратной матрицы является сумма ее положительных элементов. Переставляя указатели на строки заданной матрицы, расположить их в соответствии с ростом характеристик.

Размерность матрицы вводится пользователем, а значения являются случайными числами в диапазоне (–50 : 50).

Задача 8.8. Написать функцию, которая увеличивает размер двумерной матрицы, добавляя в нее строку. Проиллюстрировать работу этой функции:

- заполнить динамический двумерный массив случайными числами (размерность матрицы запросить у пользователя);
- запросить номер строки, которая будет вставлена;
- вставить строку с помощью созданной функции;
- заполнить эту строку нулями;
- распечатать преобразованную матрицу.

Задача 8.9. Написать функцию, которая увеличивает размер двумерной матрицы, добавляя в нее столбец. Проиллюстрировать работу этой функции:

- заполнить динамический двумерный массив случайными числами (размерность матрицы запросить у пользователя);
- запросить номер столбца, который будет вставлен;
- вставить столбец с помощью созданной функции;
- заполнить этот столбец нулями;
- распечатать преобразованную матрицу.

Задача 8.10. Написать функцию, которая удаляет строку из двумерной матрицы. Проиллюстрировать работу этой функции:

- заполнить динамический двумерный массив случайными числами (размерность матрицы запросить у пользователя);
- запросить номер строки, которая будет удалена;
- удалить строку с помощью созданной функции;
- распечатать преобразованную матрицу.

Задача 8.11. Транспонирование матрицы – это операция, после которой столбцы прежней матрицы становятся строками, а строки столбцами. Напишите функцию транспонирования матрицы.

ТЕМА 9. СТРОКИ

9.1. Строки в стиле языка С

В языке С нет специального типа «строка». Строка рассматривается как одномерный массив символов (типа char). При этом текущее содержимое строки должно оканчиваться нулевым байтом (рисунок 9.1). Нулевому байту соответствует специальный символ '\0', называемый также «нуль-терминатор».

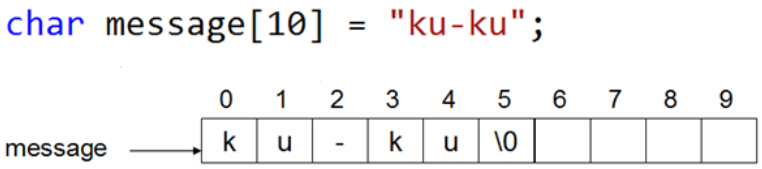


Рисунок 9.1 – **Пример объявления строки и выделенной под нее памяти**

При объявлении строки нужно учитывать необходимость размещения нулевого байта. Поэтому, если длина строки должна составлять k символов, то следует объявить массив из $k + 1$ элемента.

Например, объявление `char message[10]` предполагает, что строка может содержать не более 9 значащих символов.

При объявлении строку можно инициализировать строковым литералом:

```
char privet[]="Hello!";
```

При объявлении этой строки выделено 7 байт: 6 байт под символы и 1 байт под нуль-терминатор, который компилятор добавит автоматически.

Можно также строку инициализировать в виде списка значений символов. В этом случае нуль-терминатор нужно явно указать:

```
char privet[]={'H','e','l','l','o','!','\0'};
```

Для объявления и инициализации строки можно также использовать указатель:

```
char *privet="Hello!";
```

Эта запись означает, что в памяти выделяется место для строкового литерала (нулевой байт добавляется автоматически), а указатель privet получает значение адреса памяти, начиная с которого этот литерал расположен. Отличие от предыдущего варианта объявления в том, что теперь идентификатор privet означает переменную, которая может поменять свое значение. Например, так:

```
char *privet="Hello!";
cout<<privet<<"\n";
privet="By-by!";
cout<<privet<<"\n";
```

Этот код не означает, что новый литерал `"By-by!"` будет записан на место старого литерала `"Hello!"`: обе эти константы останутся в памяти на своих местах, но указатель privet теперь содержит другой адрес (рисунок 9.2).

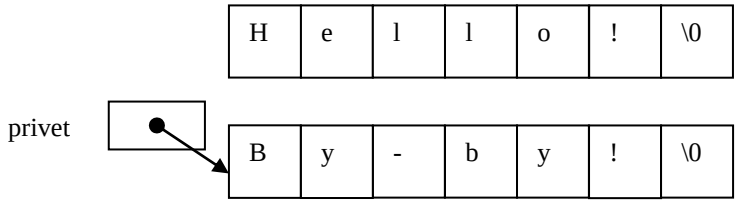


Рисунок 9.2 – **После операции присваивания указателю адреса другого литерала**

9.2. Ввод и вывод строк и символов

Если массив символов предназначен для хранения строки, которая может изменяться (например, которая будет введена с консоли), то нужно предусмотреть достаточно памяти для размещения символов. Обычно стараются выделить память с запасом (например, для ввода имени можно предусмотреть 128 символов):

```
const int MAXLEN=128;
char name[MAXLEN];
```

Для *ввода* строки можно воспользоваться потоковым вводом C++:

```
cin>>name;
```

Либо использовать функцию scanf() со спецификатором формата %s. Поскольку имя массива – это и есть адрес его начала, то знак & перед идентификатором name указывать не нужно:

```
scanf("%s",name);
```

И потоковый ввод, и функция scanf() вводят строку до первого пробела. Поэтому их нельзя использовать для ввода, например, предложения из нескольких слов.

Функция gets() осуществляет ввод строки, которая может содержать пробелы. Ввод завершается нажатием клавиши Enter:

```
gets(name);
```

Эта функция (также, как и функция scanf()) содержится в библиотеке stdio.h, которую Visual Studio подключает автоматически.

Для ввода отдельного символа с помощью функции scanf() нужно использовать спецификатор формата %c:

```
char sim;
scanf("%c",&sim);
```

Также символ можно ввести с помощью потокового ввода либо использовать функцию getchar(). Для ввода символа этим способом пользователь набирает символ на клавиатуре, а затем нажимает Enter:

```
cin>>sim; //потоковый ввод
sim=getchar(); //ввести символ и нажать Enter
```

Прототип функции getch() описан в заголовочном файле <conio.h>. Эта функция вводит символ сразу же после нажатия клавиши на клавиатуре (т. е. не нужно нажимать Enter). Часто эта функция используется для задержки экрана (например, вместо system("pause")).

```
sim=getch(); //ввести только символ
```

Visual Studio 2013 (и более поздних версий) требует для ввода использовать более безопасного вариант scanf_s(). Это требование в принципе можно отключить, задав в начале программы директиву препроцессора `#define _CRT_SECURE_NO_WARNINGS`. Однако лучше все же использовать функцию scanf_s(), поскольку для ввода строк она обеспечивает дополнительную безопасность.

Функция scanf_s() имеет третий параметр – размер буфера. Если считываемая строка больше, чем буфер, то ничего не будет введено. Размер буфера нужно рассчитать так, чтобы поместился и нуль-терминатор. Таким образом, функция scanf_s() осуществляет контроль того, чтобы при вводе данные не были записаны в область памяти, которая для них не предназначена. Аналогично работает и функция gets_s().

Примеры использования функций scanf_s и gets_s:

```
const int SIZE=20;
char str[SIZE];
scanf_s("%s",str, SIZE);
gets_s(str,SIZE);
scanf_s("%c",&symv,1);
```

Преимущества использования функции scanf_s можно продемонстрировать на следующем примере. Пусть имеется код:

```
void main()
{
     setlocale(LC_ALL,"rus");
     const int MAXLEN=4;
     char name[MAXLEN];
     puts("Как Вас зовут?");
     scanf("%s",name);
     printf("Привет, %s\n",name);
     system("pause");
}
```

И допустим, в ответ на запрос программы мы ввели слишком длинное имя (Nikita, например). Тогда в момент окончания программы возникнет ошибка выполнения (ошибки, связанные с порчей памяти часто обнаруживаются не в тот момент, когда они происходят, а при завершении программы).

Если же использовать функцию scanf_s, указав размер буфера, то такой ошибки не возникнет (правда, и строка не будет введена никакая):

```
void main()
{
     setlocale(LC_ALL,"rus");
     const int MAXLEN=4;
```

```
    char name[MAXLEN];
    puts("Как Вас зовут?");
    scanf_s("%s",name,MAXLEN);
    printf("Привет, %s\n",name);
    system("pause");
}
```

При вводе строк или символов может возникнуть проблема, связанная с тем, что в буфере ввода с консоли остаются какие-либо символы, попавшие туда при предыдущем вводе. Особенно это касается символа новой строки «\n», который записывается в буфер при нажатии клавиши Enter.

Тогда при новом вводе содержимое буфера интерпретируется как новый ввод (т. е. ввод заканчивается, не успев начаться). Чтобы избежать данной ситуации, рекомендуется перед вводом символа или строки очистить буфер ввода:

```
scanf_s("%*[^\n]");
scanf_s("%*c");
```

Для вывода одной строки можно использовать функцию puts(). Она автоматически добавляет символ «\n» – переход на новую строку: `puts("Как Вас зовут?");`

Можно также воспользоваться потоковым выводом или функцией printf():

```
cout<<"Привет, "<<name<<"\n";
printf("Привет, %s\n",name);
```

Для работы с кириллицей как на ввод, так и на вывод, нужно настроить консоль на кодировку Windows 1251. Для этого подключается библиотека <Windows.h> и используются две функции из нее:

```
#include <Windows.h>
…
SetConsoleCP(1251); //на ввод
SetConsoleOutputCP(1251); //на вывод
```

При этом шрифт в консоли должен быть установлен Lucida Console.

9.3. Функции библиотеки <string.h>

Любой алгоритм работы со строками можно реализовать, рассматривая строки как обычные массивы. Однако существует обширная библиотека функций для работы со строками <string.h>. В Visual Studio она подключается автоматически. При использовании других компиляторов ее необходимо подключить командой препроцессора `#include <string.h>`.

Перечислим основные функции работы со строками из этой библиотеки (а также из некоторых других). Следует иметь ввиду, что данный список функций соответсвует стандарту ANSI. Для последних версий Visual Studio эти функции могут быть модифицированы, их интерфейс следует уточнять в справочной системе.

Функция **`int strlen(char* s)`** вычисляет количество значимых символов в строке ('\0' не учитывается). Например, `int n=strlen(name);`

Функция **`char* strcpy(char* dest, const char* source)`** копирует строку source в строку dest. Результат функции – указатель на результирующую строку (он равен указателю на dest). Можно результат функции проигнорировать. Аргументы этой функции должны быть строками, т. е. содержать нулевой байт.

Например:

```
char myname[10]="Nikita";
char yourname[10]="Anton";
strcpy(myname,yourname);
cout<<myname; //выведется Anton
```

Функция **char* strncpy(char* dest, const char* source, int num)** копирует не более num символов из строки source в начало строки dest. Если нулевой байт не вошел в число копируемых символов, то результат будет содержать «микс» из старого и нового содержимого строки.

Например:

```
char myname[50]="Nikita Ivanov";
char yourname[10]="Anton";
strncpy(myname,yourname,2);
cout<<myname; //выведется Ankita Ivanov
```

Безопасный способ копирования, который гарантирует, что после копирования не выйдем за пределы памяти, отведенной под результирующую строку длиной MAXLEN, выглядит так:

```
strncpy(dest,source,MAXLEN-1);
```

Функция **char* strcat(char* dest, const char* source)** объединяет строку source со строкой dest. Результат сохраняется в dest (нуль-символ добавляется в конец). Программист должен позаботиться о том, чтобы объединенная строка поместилась в память, отведенную под dest (если соседняя память будет испорчена, то возникнет ошибка времени выполнения).

Например:

```
const int MAXLEN=10;
char first[MAXLEN]="One";
char second[]="Two";
strcat(first,second);
cout<<first<<"\n"; //выводится OneTwo
```

Функция **char* strncat(char* dest, char* source, int num)** объединяет не более num символов строки source со строкой dest. Результат сохраняется в dest. Нуль-символ остается «в наследство» от строки dest (т. е. символы строки-источника фактически вставляются между последним значащим символом dest и нулевым байтом).

Приведем безопасный способ конкатенации (здесь MAXLEN – размерность массива first):

```
strncat(first,second,MAXLEN-strlen(first)-1);
```

Функция **int strcmp(const char* s1, const char* s2)** сравнивает две строки в лексикографическом порядке. Возвращает отрицательное число, если *s*1 < *s*2, ноль, если *s*1 == *s*2 и положительное число, если *s*1 > *s*2.

При сравнении двух строк они анализируются посимвольно. Если соответствующие символы строк равны, то сравнивается следующая пара. Если строка закончилась раньше, то она считается меньше. Если код символа из пары меньше, то вся строка считается меньше. Коды букв упорядочены по алфавиту (т. е. 'A' < 'B' < 'C' и т. д.). Аналогично упорядочены коды цифр.

Например:

```
char s1[]="One";
char s2[]="One day";
if(strcmp(s1,s2)<0)
     cout<<"Первая строка меньше\n"; //выводится это сообщение
else
     if(strcmp(s1,s2)>0)
          cout<<"Первая строка больше\n";
     else
          cout<<"Строки равны\n";
```

Если задать `char s1[]="One way";char s2[]="One day";` то выведется сообщение «Первая строка больше». Если же `char s1[]="One";char s2[]="One";` то выведется сообщение «Строки равны».

Функция **int atoi(char* s)** преобразует строку *s* в число типа int. Возвращает значение или нуль, если число преобразовать нельзя. Прототип этой функции находится в заголовочном файле <stdlib.h>

Функция **long atol(char* s)** преобразует строку *s* в число типа long. Возвращает значение или нуль, если число преобразовать нельзя. Также описана в <stdlib.h>.

Функция **double atof(char* s)** преобразует строку *s* в вещественное число типа double. Возвращает значение или нуль, если число преобразовать нельзя. В строке дробная часть должна отделяться запятой. Прототип содержится в заголовочном файле <math.h>.

Функция **char* itoa(int value, char* s, int radix)** преобразует целое число value в строку *s*. Значение radix – основание системы счисления, используемое при преобразовании (от 2 до 36). Возвращает указатель на результирующую строку. Функция описана в файле <stdlib.h>.

Например:

```
int i=45;
char ss[10];
itoa(i,ss,10);
cout<<ss<<"\n"; //выводится 45
int j=atoi(ss);
cout<<j<<"\n"; //выводится 45
char s2[]="34,67";
double x=atof(s2);
cout<<x<<"\n"; //выводится 34.67
```

Функция **char* strchr(const char* s, char c)** ищет в строке *s* первое вхождение символа *c*, начиная с начала строки. В случае успеха возвращает указатель на найденный символ, иначе – возвращает NULL.

Например:

```
char mystring[]="Властелин к0лец";
char c='0'; //будем искать
char *ptr=strchr(mystring, c);
puts(mystring);
cout<<"Моя прелесть на "<<ptr-mystring+1<<" месте\n";
```

Функция **char* strrchr(const char* s, char c)** аналогично предыдущему, только поиск осуществляется с конца строки.

Функция **char* strstr(const char* s1, const char* s2)** ищет в строке *s*1 подстроку *s*2. Возвращает указатель на первый символ подстроки внутри *s*1. Если подстрока отсутствует – возвращает NULL.

Например:

```
char s1[]="example.cpp";
char *p=strstr(s1,".cpp");
if(p!=NULL)
     cout<<"Это программа на C++\n";
else
     cout<<"Это не программа на C++\n";
```

Функция **char* strlwr(char* s)** преобразует в строке *s* все прописные (большие) буквы в строчные (малые).

Функция **char* strupr (char* s)** преобразует в строке *s* все строчные (малые) буквы в прописные (большие).

Например:

```
char filename[128];
cout<<"Введите имя файла: ";
cin>>filename;
strupr(filename);
cout<<filename<<"\n"; //выводятся все буквы прописные
```

Функция **char* strtok(char* s1, const char* s2)** делит исходную строку s1 на лексемы (подстроки), разделенные одним или несколькими символами из строки s2. Компилятор Visual Studio 2013 и более поздних версий требует использовать функцию strtok_s, которая имеет следующий интерфейс:

```
char *strtok_s(char *strToken, const char *strDelimit, char **context);
```

При первом вызове функции параметр strToken – это указатель на начало разбираемой строки, а при последующих вызовах на место этого параметра передается NULL. Второй параметр strDelimit – строка, содержащая возможные разделители. Третий параметр contex – адрес вспомогательного указателя, который будет установлен на начало оставшейся части строки:

```
const int MAXLEN = 30;
char str[MAXLEN] = "mama myla ramy ";
char *ptr; //указатель на начало очередной лексемы
char *buf = NULL;//указатель на оставшуюся часть строки
//первый вызов:
ptr = strtok_s(str, " ", &buf);
cout << ptr << "\n"; //выводит mama
```

Функция вставляет нуль-терминатор после первой лексемы и возвращает указатель на ее начало (рисунок 9.3).

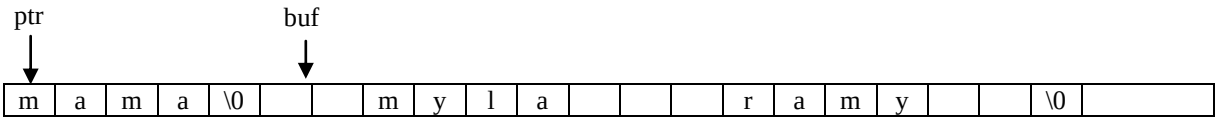


Рисунок 9.3 – **Результат первого вызова функции strtok_s**

Второй и последующий вызовы этой функции начинают анализ с символа, на который указывает последний параметр (первый параметр при этом имеет значение NULL). И после очередной лексемы опять вставляется нуль-терминатор:

```
//второй вызов:
ptr = strtok_s(NULL, " ", &buf);
cout << ptr << "\n"; //выводит myla
```

Результат выполнения второго вызова показан на рисунке 9.4.

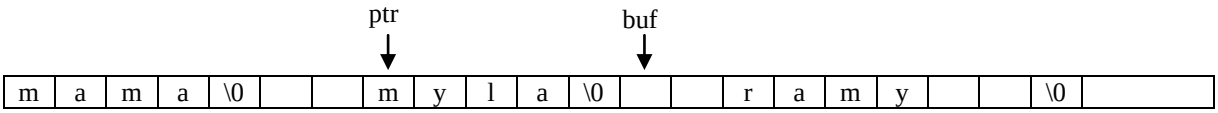


Рисунок 9.4 – **Результат второго вызова функции strtok_s**

Если при разборе больше не обнаружено лексем, то результат функции – значение NULL.

Недостатком этой функции является то, что она «портит» исходную строку, изменяя ее символы.

Пример 9.1. Программа вводит строку с пробелами функцией gets. В строке первое слово содержит фамилию студента, второе слово – его имя, а третье – средний балл. Нужно выполнить парсинг (разбор) строки и записать фамилию в символьный массив fam, имя – в символьный массив name, а средний балл – в вещественную переменную ball. Эти значения потом по отдельности выводятся на консоль:

```
#include <iostream>
using namespace std;
#include <Windows.h>
void main() {
```

```
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
    const int N = 128;//максимальная длина строки
    char str[N]; //строка-буфер для чтения строки с консоли
    char fam[N], name[N]; //строки для фамилии и имени
    double ball; //переменная для ввода среднего балла
    char * word, *next; //вспомогат. указатели при разборе строки
    //чтение строки с консоли
    puts("Введите данные студента через пробел:");
    gets_s(str);
    //разбор строки
    word = strtok_s(str, " ", &next);
    strcpy_s(fam, word);
    word = strtok_s(NULL, " ", &next);
    strcpy_s(name, word);
    word = strtok_s(NULL, " ", &next);
    ball = atof(word); //преобразование строки в вещественное число
    //вывод элементов записи на консоль
    printf("Фамилия: %s\n", fam);
    printf("Имя: %s\n", name);
    printf("Средний балл: %3.1f\n", ball);
    system("pause");
}
```

Результат работы программы показан на рисунке 9.5.

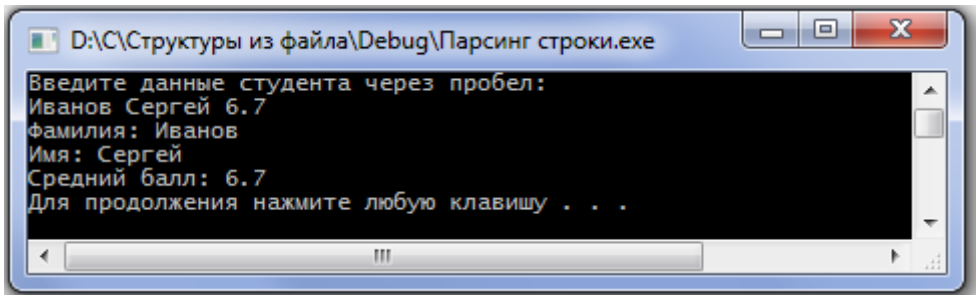


Рисунок 9.5 – **Результат парсинга строки**

Перечислим остальные функции для работы со строками:

- `int stricmp(const char* s1, const char* s2)` – аналогично функции сравнения строк strcmp(), только сравнение осуществляется без учета регистра символов.
- `int strncmp(const char* s1, const char* s2, int maxlen)` – аналогично предыдущей, только сравниваются первые maxlen символов.
- `int strnicmp(const char* s1, const char* s2, int maxlen)` – аналогично предыдущей, только сравниваются первые maxlen символов без учета регистра.
- `int strcspn(const char* s1, const char* s2)` – возвращает длину максимальной начальной подстроки строки s1, не содержащей символов из второй строки *s*2.
- `char* strnset(char* s, int c, int n)` – заполняет строку *s* символами *c*. Параметр *n* задает количество размещаемых символов в строке.
- `char* strpbrk(const char* s1, const char* s2)` – ищет в строке *s*1 первое вхождение любого символа из строки *s*2. Возвращает указатель на первый найденный символ или нуль, если символ не найден.
- `char* strrev(char* s)` – изменяет порядок следования символов в строке на обратный (кроме завершающего нулевого символа). Функция возвращает строку *s*.
- `char* strset(char* s, int c)` – заменяет все символы строки *s* заданным символом *c*.
- `int strspn(const char* s1, const char* s2)` – вычисляет длину максимальной начальной подстроки строки *s*1, содержащей только символы из строки *s*2.

Подробное описание их работы можно найти в различных справочных пособиях.

9.4. Алгоритмы работы со строками

Приведем несколько примеров работы со строками.

Пример 9.2. Необходимо написать собственную функцию копирования строк. Протестировать ее работу.

```
#include <iostream>
using namespace std;
void mycopy(char* dest, char* source)
{
     while(*source!='\0') //пока не конец строки-источника
          *dest++=*source++; //копируем символ и продвигаем указатели
     *dest='\0'; //запись нулевого байта в конец
}
void main()
{
     setlocale(LC_ALL,"rus");
     char first[15]="Первый";
     char second[]="Второй";
     printf("%s %s\n",first,second); //выводит Первый Второй
     mycopy(first,second);
     printf("%s %s\n",first,second); //выводит Второй Второй
     system("pause");
}
```

Пример 9.3. Следует ввести имя файла с расширением и проверить, является ли файл текстовым. Если это так, то вывести отдельно имя файла (без расширения).

```
#include <iostream>
using namespace std;
void main()
{
     setlocale(LC_ALL,"rus");
     const int maxlen=128;
     char filename[maxlen],name[maxlen]="";
     puts("Введите имя файла");
     gets(filename);
     strupr(filename); //меняет все буквы на прописные
     char *ptr=strstr(filename,".TXT"); //находит начало подстроки
     if(ptr!=NULL)
     {
          cout<<"Текстовый файл\n";
          strncpy(name,filename,ptr-filename); //выделяет имя до точки
          cout<<"С именем "<<name<<"\n";
     }
     else
          cout<<"Это не текстовый файл!\n";
     system("pause");
}
```

Пример 9.4. Требуется ввести строку и символ. Удалить из строки все вхождения данного символа.

```
//функция удаляет символ в строке str, на который указывает ptr
void strdel(char* str,char* ptr)
{    while(*ptr!='\0')
     {    *ptr=*(ptr+1); //сдвигаем символ влево
          ptr++; //продвигаем указатель
     }
}
```

```
void main()
{    setlocale(LC_ALL,"rus");
     const int MAXLEN=128;
     char mystring[MAXLEN];
     char sim, *ptr;
     cout<<"Введите строку: ";
     gets(mystring);
     fflush(stdin);
     cout<<"Введите символ: ";
     sim=getchar();
     while((ptr=strchr(mystring,sim))!=NULL)//ищем sim в строке
          strdel(mystring,ptr); //удаляем символ в этой позиции
     puts("Результат:");
     puts(mystring);
     system("pause");
}
```

Пример 9.5. Необходимо ввести строку и символ. Удвоить все вхождения данного символа в строке.

```
#include <iostream>
using namespace std;
//функция вставляет символ c в строку str на место ptr
void insert(char* str, char c, char* ptr)
{
     char* q=str+strlen(str); // на последний символ '\0'
     for(;q>=ptr;q--) //подвигаем указатель влево
          *(q+1)=*q; //сдвиг символа вправо
     *ptr=c; //вставка символа
}

void main()
{    setlocale(LC_ALL,"rus");
     const int MAXLEN=128;
     char mystring[MAXLEN];
     char sim, *ptr;
     cout<<"Введите строку: ";
     gets(mystring);
     fflush(stdin); // очистка буфера консоли
     cout<<"Введите символ: ";
     sim=getchar();
     ptr=mystring;
     while(*ptr!='\0')
     {
          if(*ptr==sim)
               insert(mystring,sim, ++ptr); //вставка в след. позицию
          ptr++; //переход к следующему символу
     }
     puts("Результат:");
     puts(mystring);
     system("pause");
}
```

Пример 9.6. Следует ввести строку, состоящую из слов, разделенных одним или несколькими пробелами. Пробелы могут быть и в самом начале, и в самом конце предложения. Подсчитать количество слов.

```
#include <iostream>
using namespace std;
void main()
{
```

```
     const int MAXLEN=128;
     char message[MAXLEN];
     setlocale(LC_ALL,"rus");
     puts("Введите предложение: ");
     gets(message);
     int k=0; //счетчик числа слов
     char *ptr=message; //указатель на текущий символ
     while(*ptr!='\0')
     {
          while(*ptr==' ') ptr++; //пропустить пробелы перед словом
          if(*ptr!='\0') //есть начало слова
               k++;
          while(*ptr!=' '&&*ptr!='\0') //пока не конец слова
               ptr++;
     }
     cout<<"Количество слов= "<<k<<"\n";
     system("pause");
}
```

Пример 9.7. Требуется создать список группы (количество студентов запросить у пользователя). Для каждого студента вводится фамилия, имя и отчество. Следует соединить их в одну строку, которая должна занимать ровно столько байт, сколько нужно. Создается массив указателей на размещаемые строки. Следует отсортировать список по алфавиту путем перестановки этих указателей.

Идея алгоритма состоит в том, чтобы данные очередного человека ввести сначала в буферные переменные заведомо большого размера. Потом необходимо подсчитать действительное количество памяти, которое требуется для размещения записи, и выделить память в куче.

Создается динамический массив указателей на начало строк. По мере размещения записей эти указатели получают конкретное значение. При сортировке переставляются не строки, а указатели на них.

Схема размещения данных в памяти показана на рисунке 9.6.

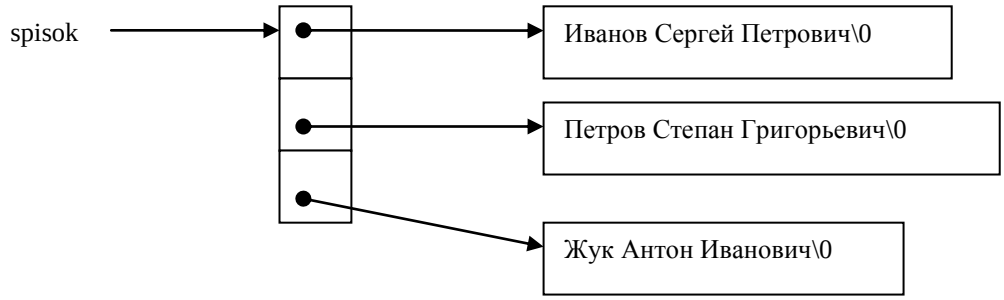


Рисунок 9.6 – **Схема размещения данных в динамической памяти**

```
#include <iostream>
#include <Windows.h>
using namespace std;
//функция печати списка
void showlist(char** spisok, int kol)
{
     for(int i=0;i<kol;i++)
          cout<<spisok[i]<<"\n";
}
//функция сортировки списка
void sort(char** spisok, int kol)
{    //метод выбора
     for(int k=0;k<kol-1;k++)
     {
          int min=k;
          for(int i=k+1;i<kol;i++)
               if(strcmp(spisok[i],spisok[min])<0)//сравниваем строки
```

```
                    min=i;
            char *temp=spisok[k];//переставляем указатели
            spisok[k]=spisok[min];
            spisok[min]=temp;
      }

}
void main()
{
      SetConsoleCP(1251); //на ввод кириллицы
      SetConsoleOutputCP(1251); //на вывод кириллицы
      const int MAXLEN=128;
      //буферные переменные:
      char name[MAXLEN],secondName[MAXLEN], fam[MAXLEN];
      int kol;
      cout<<"Введите количество людей в списке: ";
      cin>>kol;
      char **spisok=new char*[kol]; //создание массива указателей
      for(int i=0;i<kol;i++)
      {
            cout<<"Запись № "<<i+1<<"\n";
            cout<<"Введите фамилию: ";
            cin>>fam;
            cout<<"Введите имя: ";
            cin>>name;
            cout<<"Введите отчество: ";
            cin>>secondName;
            //расчет нужного количества байт
            int len=strlen(fam)+strlen(name)+strlen(secondName)+3;
            spisok[i]=new char[len];//выделение памяти под объед. строку
            //соединение фамилии и имени
            strcat(strcat(strcpy(spisok[i],fam)," "),name);
            //присоединение отчества
            strcat(strcat(spisok[i]," "),secondName);
      }
      cout<<"Список исходный:\n";
      showlist(spisok, kol);
      sort(spisok, kol);
      cout<<"Список отсортированный:\n";
      showlist(spisok, kol);
      for(int i=0;i<kol;i++) //освобождение памяти
            delete spisok[i];
      delete spisok;
      system("pause");
}
```

Задачи

Задача 9.1. Введите строку. Замените каждый символ «;» на пару символов «.,».

Задача 9.2. Введите строку. Замените каждую пару символов «.,» на символ «;».

Задача 9.3. Напишите свою версию функции strcmp(s, t). Она сравнивает символьные строки *s* и *t*, возвращая отрицательное, нулевое или положительное значение, если строка *s* соответственно меньше, равна или больше, чем строка *t*.

Задача 9.4. Напишите свою версию функции strend(s,t), которая бы возвращала 1, если строка *t* присутствует в конце строки *s*, и 0 – в противном случае.

Задача 9.5. Вводится строка, которая состоит из слов, разделенных одним или несколькими пробелами. Пробелы могут быть также перед первым и после последнего слова. Отредактировать строку: удалить все пробелы перед первым словом и после последнего слова. Между словами оставить ровно один пробел.

Для самостоятельного решения

Задача 9.6. Вводится строка, которая состоит из слов, разделенных одним или несколькими пробелами. Пробелы могут быть также перед первым и после последнего слова. Найти и распечатать все слова, начинающиеся на букву «s».

Задача 9.7. Вводится строка, которая состоит из слов, разделенных одним пробелом. Удалить из предложения второе слово.

Задача 9.8. Вводится строка, которая состоит из слов, разделенных одним или несколькими пробелами. Пробелы могут быть также перед первым и после последнего слова. Найти и распечатать самое длинное слово.

Задача 9.9. Вводится строка, которая состоит из слов, разделенных одним пробелом. Поменять местами первое и последнее слово.

Задача 9.10. Необходимо написать приложение «Телефонный справочник». Пользователь вводит фамилию и телефон, которые потом объединяются в одну строку. Данные должны занимать минимальное количество памяти. Реализовать меню с возможностью добавления, удаления, просмотра и сортировки записей.

Задача 9.11. Написать функцию, которая определяет, является ли строка палиндромом (т. е. строкой, которую можно читать и слева направо, и справа налево: «А роза упала на лапу Азора», «Аргентина манит негра», «Я ем змея»). Рекомендуется: из строки предварительно удалить пробелы и привести ее к одному регистру.

ТЕМА 10. СТРУКТУРЫ

10.1. Структура – пользовательский тип данных

Помимо стандартных типов данных программист может создавать свои собственные составные типы. С одним из них мы уже знакомы – это перечисляемый тип enum. В этом модуле познакомимся еще с одним составным типом, который очень активно используется в программировании.

Структура – это сложный тип данных, который является объединением нескольких объектов разного типа (для сравнения: массив – объединение объектов одного типа).

Элементами структуры могут быть переменные, массивы, указатели, а также другие структуры.

Объявляя структуру, программист может ввести собственное имя типа, которое затем используется для описания переменных. Важно понимать, что при описании переменных выделяется память, а при описании типа – нет.

Для описания структуры используется ключевое слово struct, после которого указывается имя создаваемого типа, а затем в фигурных скобках перечисляются элементы (поля) структуры: тип и имя каждого поля. После описания типа ставится точка с запятой:

```
struct purchase
{
     char     goodName[30]; //имя товара
     int      number;     //количество
     double   price;   //цена за единицу
};   //объявлен тип “закупка”, память не выделяется
```

При описании переменной указывается слово stuct, имя типа-структуры и имя создаваемой переменной. Переменную типа «структура» можно инициализировать списком в фигурных скобках:

```
struct purchase myPurchase1={"milk",3,9600};
//объявлена и инициализирована переменная
//выделяется память в стеке
```

По стандарту ANSI при описании переменной нужно указывать ключевое слово struct, но компилятор Visual Studio позволяет его опустить. То есть представленное выше объявление переменной можно записать и так:

```
purchase      myPurchase1={"milk",3,9600};
```

Имя типа можно не вводить, если сразу объявляются все нужные переменные и далее в программе этот тип больше не употребляется:

```
struct
{
     int day;
     int month;
} mybirthday, hisbirthday;
```

Обращение к элементам структуры осуществляется через операцию «точка»: указывается имя переменной-структуры, и после точки – имя поля:

```
cout<< myPurchase1.goodName<<"\n";
```

Элемент структуры может иметь любой тип, в том числе быть другой структурой. Таким образом, структуры могут быть вложены друг в друга:

```
struct date
{
     int      day;
     int      month;
     int      year;
};
struct student
{
     char     name[40]; //имя
     char     secondName[40]; //фамилия
     date     birthDate; //дата рождения
} ivanov;            //объявление типа student
//и переменной этого типа ivanov
```

Однако самовложение структур запрещено. Например, нельзя в структуре student описать поле типа student.

Операция доступа к элементам структуры (точка) вычисляется слева направо:

```
ivanov.birthDate.year=1988;
//в переменной ivanov элемент birthDate и в нем элемент year
```

С элементом структуры можно работать как с обычной переменной: присваивать ей значение, выводить на консоль и т. д.

Пример 10.1. Демонстрация использования структуры «студент»: инициализация структуры с помощью списка, затем изменение даты рождения с помощью операции присваивания и вывод данных о студенте на консоль:

```
#include <iostream>
using namespace std;
void main()
{
     struct date
     {
          int      day;
          int      month;
          int      year;
     };
     struct student
     {
          char    name[40]; //имя
          char    secondName[40]; //фамилия
          date    birthDate; //дата рождения
     };      //объявление типа student
     //объявление и инициализация переменной ivanov типа student
     student ivanov={"Сергей","Иванов",{3,11,1988}};
     setlocale(LC_ALL,"rus");
     ivanov.birthDate.day=8; //изменение даты рождения
     cout<<"Информация о студенте:\n";
     cout<<ivanov.secondName<<" "<<ivanov.name<<"\n";
     cout<<"Дата рождения: "<<ivanov.birthDate.day;
     cout<<"."<<ivanov.birthDate.month;
     cout<<"."<<ivanov.birthDate.year<<"\n";
     system("pause");
}
```

Переменной типа «структура» можно присвоить значение другой переменной этого же типа. При этом выполняется копирование элементов одной структуры в другую:

```
purchase myPurchase1={"milk",3,9600};
purchase myPurchase2;
myPurchase2=myPurchase1;
```

Чтобы использовать тип структуры как параметр или результат функции, он должен быть объявлен глобально. В отличие от имени массива, имя структуры не является адресом. Поэтому в функцию структура передается по значению (создается копия переменной-структуры). Если же функция должна изменять значения элементов структуры, можно воспользоваться ссылкой или указателем на структуру:

```
#include <iostream>
using namespace std;
//глобальное объявление типа-структуры
struct purchase
{    char goodName[30];
     int number;
     double price;
};
//функция печати элементов структуры
//структура копируется в локальную переменную a
void print (purchase a)
{    cout<<"Наименование товара: "<<a.goodName<<"\n";
     cout<<"Количество: "<<a.number<<"\n";
     cout<<"Цена: "<<a.price<<"\n";
}
//функция изменения названия товара
//структура передается по ссылке
```

```
void changeGood (purchase &a, char *string)
{    strncpy(a.goodName,string,sizeof(a.goodName)-1);
     a.goodName[sizeof(a.goodName)]='\0';
}
void main()
{
     setlocale(LC_ALL,"rus");
     //объявление инициализация переменной-структуры
     purchase zakup1={"Milk",10, 9600};
     print(zakup1); //вызов функции печати
     changeGood(zakup1,"Shugar"); //вызов функции изменения товара
     print(zakup1); //снова печать структуры
     system("pause");
}
```

В реализации функции changeGood() используется функция безопасного копирования srtncpy(), которая копирует ровно столько символов строки, сколько может разместиться в элементе goodName. Функция sizeof() возвращает количество байт памяти, выделенное под элемент goodName.

10.2. Указатель на структуру. Размещение структуры в памяти

Можно объявить указатель на структуру и затем присвоить ему адрес конкретной переменной:

```
purchase      myPurchase1={"milk",3,9600};
purchase *ptr; //указатель на структуру "закупка"
ptr=&myPurchase1; //операция взятия адреса структуры
```

Через указатель можно обратиться к элементам структуры двумя способами:

- используя операцию разыменования (*):

```
cout<<"Имя товара: "<<(*ptr).goodName<<"\n";
```

- используя операцию обращения к члену структуры по указателю(->):

```
cout<<"Имя товара: "<<ptr->goodName<<"\n";
```

Пример 10.2. Перепишем функцию changeGood() из предыдущего примера с использованием указателя вместо ссылки:

```
void changeGood (purchase *a, char *string)
{    strncpy(a->goodName,string,sizeof(a->goodName)-1);
     a->goodName[sizeof(a->goodName)]='\0';
}
```

Вызов такой функции будет выглядеть так:

```
changeGood(&zakup1,"Shugar");
```

При использовании функции sizeof() применительно к структуре, нужно учитывать следующее: размер структуры в памяти не равен сумме размеров составляющих ее элементов. Каждый элемент в памяти выравнивается по границе, кратной его размеру. Например, int (4 байта) – адрес кратен четырем.

Рассмотрим следующий пример:

```
struct foo
{
```

```
    char ch;
    int value;
};
```

Результат функции sizeof(foo) равен 8 байтам. Распределение памяти показано на рисунке 10.1.

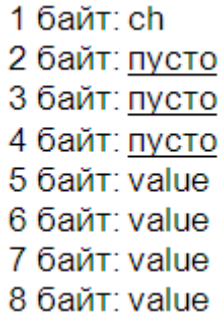


Рисунок 10.1 – **Распределение памяти для структуры типа foo**

10.3. Массивы структур

Структуры, как и другие переменные, можно объединить в массивы. Обращение к элементу структуры, которая включена в массив, имеет следующий формат:

```
имя_массива[индекс].имя_элемента
```

Пример 10.3. Создадим массив из 4 элементов. Каждый из элементов массива является структурой, содержащей информацию об определенном студенте:

```
#include <iostream>
#include <Windows.h>
using namespace std;
struct student
{    char      name[40];      //имя
     int       age;           //возраст
};
const int N=4;
void main()
{
     SetConsoleCP(1251);
     SetConsoleOutputCP(1251);
     //объявление и инициализация массива структур
     student gruppa[N]={{"Иванов",28},{"Семенов",25},
     {"Петров",20},{"Кулагин",24}};
     cout<<"Исходный список группы"<<"\n";
     for(int i=0;i<N;i++)
          cout<<gruppa[i].name<<" "<<gruppa[i].age<<"\n";
     cout<<"Введите фамилию студента №2: ";
     cin>>gruppa[1].name; //ввод данных в элемент структуры
     cout<<"Введите возраст студента №2: ";
     cin>>gruppa[1].age;
     cout<<"Измененный список группы"<<"\n";
     for(int i=0;i<N;i++)
          cout<<gruppa[i].name<<" "<<gruppa[i].age<<"\n";
     system("pause");
}
```

Пример результата работы этой программы показан на рисунке 10.2.

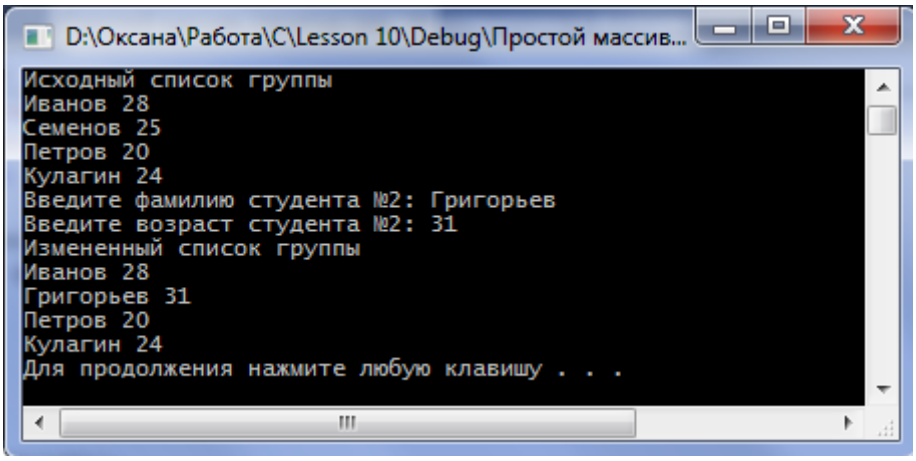
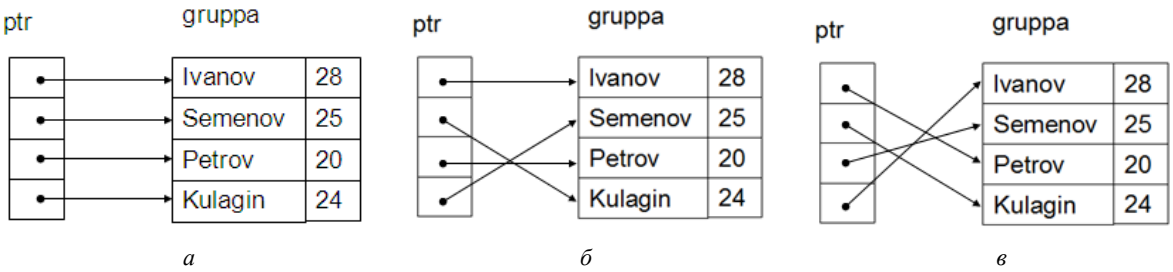


Рисунок 10.2 – **Результат работы программы вывода массива структур**

Если нужно отсортировать массив структур, то обычно сами элементы не перемещают, так как они достаточно громоздки (требуется повышенное количество времени и памяти для их копирования). Вместо этого создается вспомогательный массив указателей, которые и переставляют при сортировке. Этот способ хорош еще и тем, что можно выполнять сортировку по различным ключам. На рисунке 10.3 показано состояние массива указателей до сортировки, после сортировки по фамилии и после сортировки по возрасту.



а *б* *в*

Рисунок 10.3 – **Сортировка массива структур: а) исходный массив указателей, б) после сортировки по фамилии, в) после сортировки по возрасту**

Пример 10.4. Написать программу создания списка студентов и его печати:
- в виде не отсортированного списка (как было введено);
- в виде списка, отсортированного по имени;
- в виде списка, отсортированного по возрасту.

Использовать меню.

При решении для простоты будем использовать статический массив и меню на основе оператора switch. Разобрав этот вариант, попытайтесь его модифицировать, используя динамический массив и меню на основе указателей на функцию.

```
#include <iostream>
#include <Windows.h>
using namespace std;
struct student
{    char     name[40];     //имя
     int      age;          //возраст
};
const int N=6;
//функция сравнивает две структуры по элементу, определяемому
//параметром prizn: если prizn=1, то сравнение по имени студента, а
//если равен 0 - то по возрасту.
//Результат >0, если первый аргумент больше, <0,если меньше
// и =0 при равенстве
int compare (student *a, student *b, int prizn)
{    if (prizn)
```

```
          return strcmp(a->name,b->name); //сравнение строк
     else
          if(a->age>b->age) return 1;
          else if(a->age<b->age) return -1;
               else return 0;
}
//сортировка методом пузырька с использованием массива указателей по
//элементу, определяемому параметром prizn
void sortirovka (student **ptr, int n, int prizn)
{    student *temp;
     for(int k=n-1;k>=1;k--)
          for(int i=0;i<k;i++)
               if(compare(ptr[i],ptr[i+1],prizn)>0)
               {    temp=ptr[i]; ptr[i]=ptr[i+1];    ptr[i+1]=temp;}
}
//печать отсортированного массива: создает вспомогательный массив
//указателей, вызывает функцию сортировки, затем выводит на консоль
//в порядке, определяемом массивом указателей
void printsort ( student a[], int n, int prizn)
{
//выделяем память под массив указателей
     student **ptr=new student *[n];
     for (int i=0;i<n;i++)      //инициализация массива указателей
          ptr[i]=&a[i];
     sortirovka(ptr,n,prizn);   //вызов функции сортировки
     //печать через массивы указателей
     for(int i=0;i<n;i++)
          cout<<ptr[i]->name<<" - "<<ptr[i]->age<<"\n";
     delete [] ptr;
}
//функция выводит текст меню и вводит вариант выбора
//возвращает число от 1 до 5
int menu()
{
     short choice;
     do
     {
          cout<<"Выберите режим работы: \n";
          cout<<"1 - ввод списка студентов\n";
          cout<<"2 - печать списка студентов без сортировки\n";
          cout<<"3 - печать списка студентов, отсорт. по имени\n";
          cout<<"4 - печать списка студентов, отсорт. по возрасту\n";
          cout<<"5 - конец работы\n";
          cout<<"Ваш выбор: ";
          cin>>choice;
          if(choice<1||choice>5)
               cout<<"Неверный выбор! Повторите ввод!\n";
     }
     while(choice<1||choice>5);
     return choice;
}
//функция вводит данные в массив студентов
void vvod(student gruppa[], int n)
{
     cout<<"Ввод данных\n";
     for(int i=0;i<n;i++)
     {
          cout<<"Студент №"<<i+1<<"\n";
          cout<<"Имя: ";
          cin>>gruppa[i].name;
          cout<<"Возраст: ";
          cin>>gruppa[i].age;
```

```
	}
}
//функция выводит массив студентов без сортировки
void show(student gruppa[],int n)
{
	cout<<"Список студентов без сортировки:\n";
	for(int i=0;i<n;i++)
	{
		cout<<i+1<<". "<<gruppa[i].name<<", "<<gruppa[i].age<<"\n";
	}
}
//инициализация массива пустыми значениями
//на случай обращения к нему до ввода данных
void init(student a[],int n)
{
	for(int i=0;i<n;i++)
	{
		a[i].name[0]='\0';
		a[i].age=0;
	}
}
void main()
{
	SetConsoleCP(1251);
	SetConsoleOutputCP(1251);
	student gruppa[N]; //массив структур
	int choice;
	init(gruppa,N); //инициализация массива
	while(true)
	{
		choice=menu();
		switch (choice)
		{
		case 1: vvod(gruppa,N);break;
		case 2: show(gruppa,N);break;
		case 3: printsort(gruppa,N,1); break;//сортировка по имени
		case 4: printsort(gruppa,N,0); break;//сортировка по возр.
		case 5: exit(0);
		}
	}
}
```

Для упрощения понимания этой программы на рисунке 10.4 приведена иерархия функций (т. е. показано, какая функция какую вызывает).

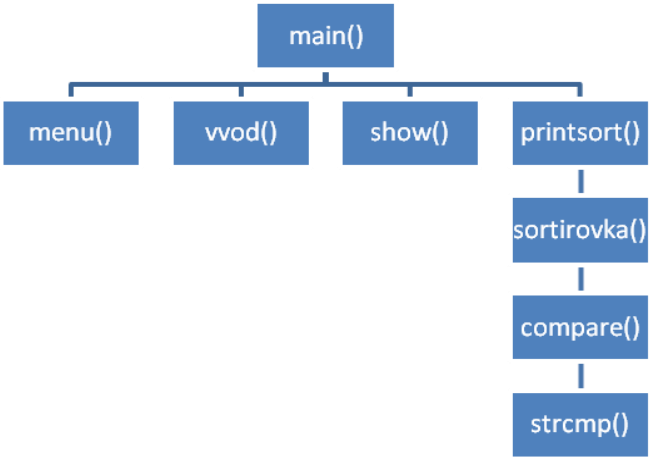


Рисунок 10.4 – **Диаграмма вызовов функций в программе**

10.4. Динамические стуктуры данных

Часто в серьезных программах надо использовать данные, количество которых должно меняться в процессе работы. Динамические массивы здесь не выручают, поскольку заранее нельзя сказать, сколько памяти надо выделить – это выясняется только в процессе работы.

Например, нужно проанализировать текст и определить, какие слова и в каком количестве в нем встречаются, причем общее количество слов заранее неизвестно.

В таких случаях применяют данные особой структуры, которые представляют собой отдельные элементы, связанные с помощью ссылок. Каждый элемент (узел) состоит из двух областей памяти: данных и ссылок (рисунок 10.5). Ссылки – это адреса других узлов этого же типа, с которыми данный элемент логически связан. В языке С для организации ссылок используются переменные-указатели.

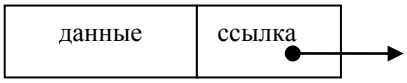


Рисунок 10.5 – **Узел динамической структуры данных**

При добавлении узла в такую структуру выделяется новый блок памяти, и с помощью ссылок устанавливаются связи этого элемента с уже существующими. Для обозначения конечного элемента в цепи используются нулевые ссылки (NULL).

В простейшем случае каждый узел имеет одну ссылку. В результате получаем структуру, называемую *линейным односвязным списком*.

Пусть для определенности узел имеет одно поле данных типа int и одну ссылку, которая является указателем на другой узел. Для описания типа такого узла используем следующую структуру:

```
struct Node
{
     int data; //данные
     Node *next; //указатель на следующий элемент того же типа
};
```

Чтобы иметь возможность описывать параметры функции этого типа, описание структуры должно быть глобальным, т. е. располагаться в начале программы, вне любой функции.

Обращение к элементу списка будет осуществляться через указатель, который в данном примере можно описать так:

```
Node* elem;
```

Для обращения к элементам структуры через этот указатель используется конструкция «стрелка»:

```
elem->data или elem->next
```

Особыми видами линейных списков являются стек, очередь и дек. Для этих структур разрешены только операции вставки и удаления первого и (или) последнего элемента.

Стек (stack) – это упорядоченный набор элементов, в котором добавление новых и удаление существующих элементов допустимо только с одного конца, который называется вершиной стека.

Стек также называют структурой типа LIFO (Last In – First Out) – последним пришел, первым ушел.

Стек похож на стопку с подносами, уложенными один на другой. Для того чтобы достать какой-то поднос, надо снять все подносы, которые лежат на нем, а положить новый поднос можно только сверху всей стопки. На рисунке 10.6 показан стек, содержащий 6 элементов.

В современных компьютерах стек используется для размещения локальных переменных, параметров функции и адреса возврата из функции.

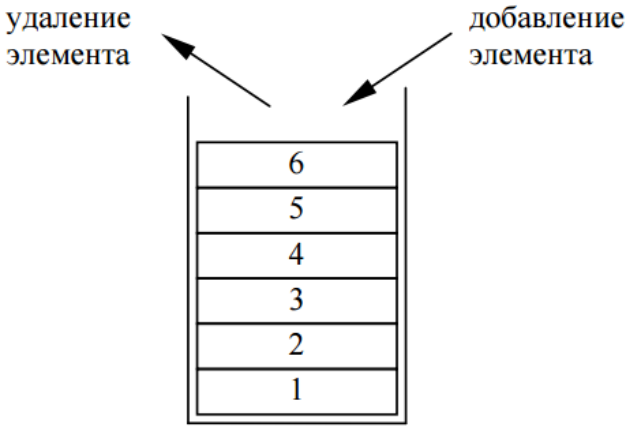


Рисунок 10.6 – **Наглядное изображение стека**

Очередь (queue) – это упорядоченный набор элементов, в котором добавление новых элементов допустимо с одного конца (он называется концом очереди), а удаление существующих элементов – только с другого конца, который называется началом очереди.

Хорошо знакомой моделью является очередь в магазине. Очередь называют структурой типа FIFO (First In – First Out) – первым пришел, первым ушел.

На рисунке 10.7 изображена очередь из 3 элементов.

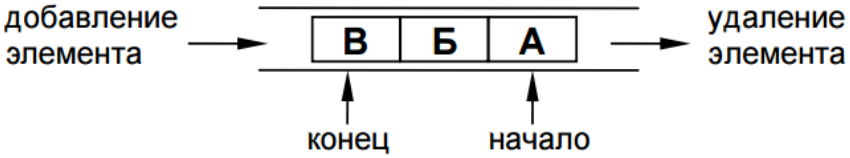


Рисунок 10.7 – **Наглядное изображение очереди**

Наиболее известные примеры применения очередей в программировании – очередь событий системы Windows и ей подобных. Очереди используются также для моделирования в задачах массового обслуживания (например, обслуживания клиентов в банке).

Дек (deque) – это упорядоченный набор элементов, в котором добавление новых и удаление существующих элементов допустимо с любого конца.

Для дека разрешены четыре операции:

- добавление элемента в начало;
- добавление элемента в конец;
- удаление элемента из начала;
- удаление элемента из конца.

Пример 10.5. *Реализация стека с помощью односвязного списка*.

Линейный список, реализующий стек, показан на рисунке 10.8. Доступ к стеку пользователь получает через указатель на его вершину (Head). Каждый элемент имеет указатель на следующий (кроме последнего, который в поле next содержит NULL).

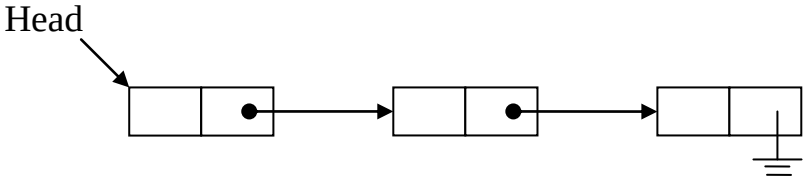


Рисунок 10.8 – **Линейный список, реализующий стек**

Для создания стека достаточно присвоить значение NULL указателю Head. Должны быть созданы следующие функции для работы со стеком:

- добавление элемента;

• удаление элемента;
• просмотр (печать) стека;
• очистка стека (удаление всех элементов).
Рассмотрим каждую функцию более подробно.

Добавление элемента

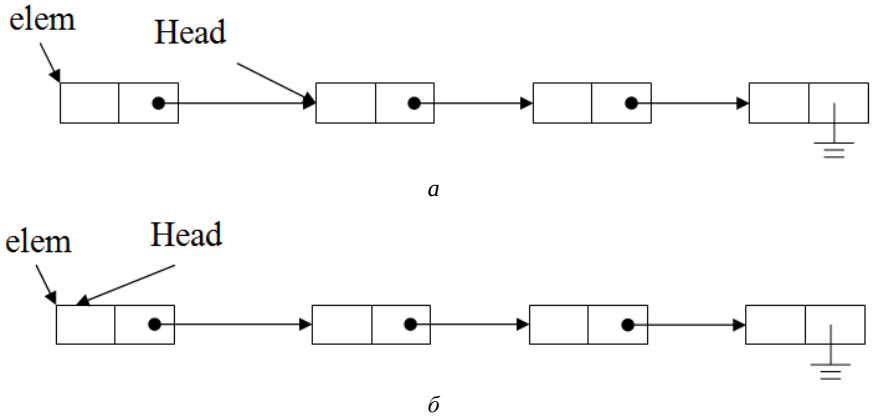
```
void AddStek(int data, Node* &Head)
{
     Node* elem=new Node; //выделение памяти под новый элемент
     elem->data=data;    //внесение данных
     elem->next=Head;    // связь с остальным списком
     Head=elem;          //голова стека теперь на новом элементе
}
```

Аргументами функции являются:
• Целое число data, представляющее собственно данные, которые нужно занести в стек.
• Указатель на вершину стека, т. е. указатель на тип Node. Обратите внимание, что указатель Head передается по ссылке. Это сделано потому, что после добавления нового элемента в стек указатель на вершину должен измениться. Использование ссылки языка C++ вместо конструкции «указатель на указатель» позволяет упростить код программы.

В тексте функции сначала выделяется память под новый узел с помощью операции new. Переменная elem получает значение указателя на эту выделенную память.

Затем заполняется поле данных нового элемента, а ссылка на следующий узел устанавливается на голову стека (рисунок 10.9а).

И, наконец, после добавления элемента указатель на голову стека изменяется, т. е. указывает теперь на новый элемент – рисунок 10.9б).



а

б

Рисунок 10.9 – **Добавление элемента в стек**

Удаление элемента

```
int GetStek(Node* &Head)
{
     if(!Head) return 0; //если стек пуст - возврат 0
     int data=Head->data; //копирование данных из вершины
     Node* tmp=Head;      //вспомогательный указатель на вершину стека
     Head=Head->next;     //голова стека сдвигается на следующий узел
     delete tmp;     //освобождение памяти из-под изъятого элемента
     return data;         //возврат данных
}
```

Аргументом функции является указатель на вершину стека (который может быть изменен внутри функции). Результат – данные, хранящиеся в вершине стека.

В самом начале функции выполняется проверка: не пуст ли стек. Затем данные копируются во вспомогательную переменную.

Вспомогательный указатель устанавливается в вершину стека. Он будет нужен для освобождения памяти. Головной указатель стека продвигается на следующий элемент, а бывший верхний узел уничтожается.

Просмотр стека

```
void ShowStek(Node* Head)
{
     cout<<"Содержимое стека: \n";
     Node* tmp=Head;
     while(tmp!=NULL) //пока не конец списка
     {
          cout<<tmp->data<<" ";  //выводится поле данных
          tmp=tmp->next;         //указатель на следующую вершину
     }
     cout<<"\n";
     }
}
```

Функция просмотра никак стек не изменяет, поэтому указатель на вершину стека передается в нее не по ссылке, а по значению.

Очистка стека

```
void ClearStek(Node* &Head)
{
     Node* Next;
     while(Head) //указатель не NULL
     {
          Next=Head->next; //запоминаем указатель на следующий элемент
          delete Head;     //удаляем элемент в вершине
          Head=Next;       //указатель вершины – на следующий
     }
}
```

Аргумент функции – указатель на вершину стека (будет изменен – установлен в NULL). Функция последовательно очищает память, заданную каждым элементом, пока указатель Head не примет нулевое значение.

Программа, которая демонстрирует использование всех этих функций, следующая:

```
#include <iostream>
using namespace std;
struct Node
{
     int data; //данные
     Node *next; //указатель на следующий элемент
};
void AddStek(int , Node* &);
void ShowStek(Node*);
void ClearStek(Node* &);
int GetStek(Node* &);
void main()
{
     setlocale(LC_ALL,"rus");
     Node *Head=0; //создание - обнуление головного указателя
     int n;
     int data;
     cout<<"Введите число элементов списка: ";
     cin>>n;
```

```
	for(int i=0;i<n;i++)
	{
		cout<<"Введите элемент "<<i+1<<" ";
		cin>>data;
		AddStek(data,Head);
	}
	ShowStek(Head); //печать элементов стека
	cout<<"Сколько элементов извлечь? ";
	cin>>n;
	for(int i=0;i<n;i++)
	{
		if(!Head)break; //если данные закончились - выйти
		data=GetStek(Head);
		cout<<"Извлечен элемент: "<<data<<"\n";
	}
	ShowStek(Head); //печать элементов стека
	ClearStek(Head); //очистка памяти – удаление стека
	system("pause");
}
```

10.5. Объединения (смеси)

Объединение – это особый вид структуры, в которой элементы накладываются друг на друга.

Объединения позволяют нескольким переменным различных типов занимать один участок памяти. Компилятор автоматически создает переменную достаточного размера для хранения наибольшей переменной, присутствующей в объединении.

В каждый момент времени активным является только один компонент объединения, а инициализировать можно только первый элемент.

При объявлении объединения используется ключевое слово union:

```
union   mymix
{
	int i;
	char ch;
} myvar;
```

В этом примере объявлен сразу тип (mymix) и переменная (myvar), под которую выделена память. Размещение элементов объединения в памяти показано на рисунке 10.11. Следует отметить, что при использовании различных компиляторов выравниваться элементы объединения могут по-разному.

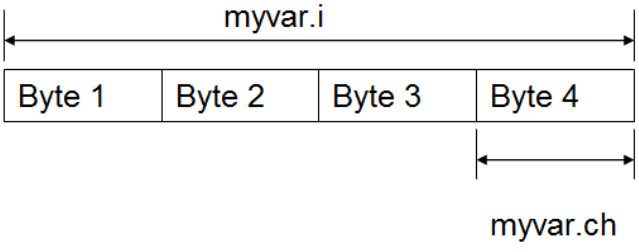


Рисунок 10.11 – **Размещение элементов объединения в памяти**

Объединения часто используются при необходимости преобразования типов. Например, в программе ниже объединение используется для преобразования числа в символ с заданным кодом:

```
#include <iostream>
using namespace std;
void main()
```

```
{
     setlocale(LC_ALL,"rus");
     union mymix
     {
          int i;
          char ch;
     } myvar;
     cout<<"Введите код символа: ";
     cin>>myvar.i;
     cout<<" Ему соответствует символ: "<<myvar.ch<<"\n";
     system("pause");
}
```

Также можно привести пример использования объединения для просмотра внутреннего представления числа. В программе ниже выводится шестнадцатеричное представление в памяти вещественного числа –1 (напомним, что тип float занимает в памяти 4 байта, как и unsigned int).

```
#include <iostream>
using namespace std;
void main()
{    setlocale(LC_ALL,"rus");
     union mymix
     {    float f;
          unsigned int u;
     } myvar;
     myvar.f=-1;
     cout<<"Вещественное число: "<<myvar.f;
     cout<<" имеет код: "<<hex<<myvar.u<<"\n";
     system("pause");
}
```

И, наконец, объединение можно использовать для экономии памяти. Например, мы хотим создать универсальный тип «геометрическая фигура». Это может быть круг (храним радиус), прямоугольник (храним две стороны) или треугольник (массив из трех сторон). Размещаем все это на одном и том же участке памяти с помощью объединения:

```
union geom_fig
{
     int radius; //круг
     int a[2]; //прямоугольник
     int b[3]; //треугольник
} myfig;
```

Очевидно, что если компонент radius получил свое значение, то нет смысла обращаться, например, к массиву *a*.

Нужно помнить, что объединения не безопасны. Поскольку они позволяют рассматривать одну и ту же область памяти как данные разных типов, объединения могут являться источником трудноуловимых ошибок. Кроме того, для корректного использования объединений нужно хорошо знать особенности конкретного компилятора языка С.

10.6. Битовые поля

Элементом структуры может быть не только переменная, но и несколько соседних двоичных разрядов внутри одного целого. Такие элементы называются полями.

Поле может иметь тип unsigned int или signed int.

К битовым полям не может применяться операция адреса, так как они находятся внутри байта (наименьшая адресуемая единица в языке С – один байт).

Описание битового поля предполагает указание после имени поля двоеточия и размера поля в битах:

```
struct myfields
{
     unsigned int a:2;
     unsigned int :3;
     unsigned int c:3;
} myvar;
```

Если имя поля не указывается, то эти биты внутри переменной не используются (скрытое поле). Размещение данных в памяти для приведенного выше примера показано на рисунке 10.12.

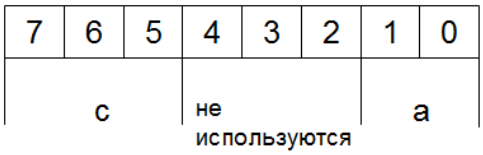


Рисунок 10.12 – **Двоичные разряды переменной myvar**

Основное назначение битовых полей – экономия памяти.

Пример 10.6. Пусть требуется хранить данные о погоде:

- скорость ветра (м/с) (максимальная скорость ветра, которая наблюдалась, была 113 м/с);
- относительная влажность воздуха в % (максимум 100);
- атмосферное давление (мм рт. ст.) (максимальное значение, которое было ранее 813 мм рт. ст.);
- температура воздуха (от –50°С до 50°С).

Если задавать все эти данные переменными типа int, то займем 4 · 4 = 16 байт. Используем битовые поля. Выделим:

- под скорость 7 бит (максимальное значение может быть 127);
- под влажность 7 бит (максимальное значение также 127);
- под давление 11 бит (максимально возможное значение 2047);
- под температуру 7 бит, со знаком (от –63 до +63).

Итого потратим 32 бита = 4 байта.

Ниже приведено описание соответствующей структуры и вывод ее размера в функции main:

```
#include <iostream>
using namespace std;
struct pogoda
{    unsigned int windRate:7;
     unsigned int humidity:7;
     unsigned int pressure:11;
     signed int temperature:7;
};
void main()
{
     setlocale (LC_ALL,"rus");
     pogoda weather;
     cout<<"Размер структуры: "<<sizeof(pogoda)<<"\n";
     system("pause");
}
```

Ввести данные сразу в битовые поля невозможно, так как нельзя взять адрес поля. Поэтому используем для ввода вспомогательную переменную. Затем ее значение присвоим полю. При присваивании произойдет преобразование данных в нужное количество бит:

```
void input(pogoda &a)
{    int temp;
     cout<<"Введите скорость ветра (м/с): ";
     cin>>temp;
     a.windRate=temp;
     cout<<"Введите влажность воздуха (%): ";
     cin>>temp;
     a.humidity=temp;
     cout<<"Введите атмосферное давление (мм рт. столба): ";
     cin>>temp;
     a.pressure=temp;
     cout<<"Введите температуру воздуха (С): ";
     cin>>temp;
     a.temperature=temp;
}
```

При выводе битовых полей нет никаких ограничений. С ними можно обращаться как с обычными элементами структуры:

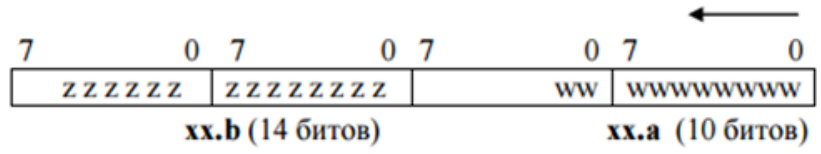
```
void print(pogoda a)
{
     cout<<"Скорость ветра: "<<a.windRate<<"\n";
     cout<<"Влажность воздуха "<<a.humidity<<"\n";
     cout<<"Атмосферное давление: "<<a.pressure<<"\n";
     cout<<"Температура воздуха: "<<a.temperature<<"\n";
}
```

Операции с битовыми полями являются машинно-зависимыми. В зависимости от используемого компьютера и операционной системы поля могут размещаться как слева направо, так и справа налево. В некоторых компьютерах поля могут пересекать границы машинного слова, в других – нет. Размещение полей с длиной, не кратной длине слова, возможно «встык» или с выравниванием.

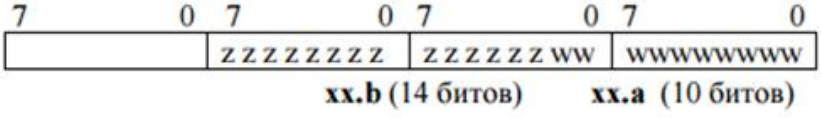
Например, для структуры вида:

```
struct
{    int a:10;
     int b:14;
}xx;
```

представление в памяти переменной *xx* может иметь вид как на рисунке 10.13а или как на рисунке 10.13б (в зависимости от реализации).



а



б

Рисунок 10.13 – **Представление в памяти переменной *xx*: а) при выравнивании на границе слова или байта; б) при плотной упаковке**

Чтобы правильно использовать битовые поля, нужно выяснить все эти нюансы применительно к своему компьютеру. Программы, использующие битовые поля, не являются переносимыми.

10.7. Битовые операции

Язык С был задуман как язык для разработки операционных систем. Поэтому в нем есть «низкоуровневые» возможности. Одна из них – это возможность выполнять побитовые операции над данными. Эти операции нельзя применять к вещественным числам. Возможны следующие битовые операции:

• ~ – инверсия.

Все биты аргумента меняются на противоположные (0 на 1, а 1 на 0).

Например:

```
unsigned char x=~89;
//значение переменной x=166
```

Пояснение:

$89_{10}=01011001_2$
$\sim 89_{10}=10100110_2=166_{10}$

• & – битовое *и*.

Для каждой пары соответствующих битов аргументов выполняется операция «и» по правилам: $1 \cdot 1 = 1$; $1 \cdot 0 = 0$; $0 \cdot 0 = 0$. То есть у результата единица будет только в том разряде, в котором у обоих аргументов стояли единицы.

Например:

```
unsigned char x=89,y=101;
unsigned char z=x&y;
//значение переменной z=65
```

Пояснение:

$x=89_{10}= 01011001_2$
$y=101_{10}= 01100101_2$
$x\&y = 01000001_2=65_{10}$

• | – битовое *или*.

Для каждой пары битов аргументов выполняется операция «или» по правилам: 1 + 1 = 1; 1 + 0 = 1; 0 + 0 = 0. То есть у результата единица будет в том разряде, где хотя бы у одного аргумента в этом разряде была единица.

Например:

```
unsigned char x=89,y=101;
unsigned char z=x|y;
//значение переменной z=125
```

Пояснение:

$x=89_{10}=01011001_2$
$y=101_{10}=01100101_2$
$x|y =01111101_2=125_{10}$

• ^ – битовое исключающее *или*.

Для каждой пары битов аргументов выполняется операция «исключающее или» по правилам: 1 + 1 = 0; 1 + 0 = 1; 0 + 0 = 0. То есть у результата будет единица в разряде, если только у одного аргумента в этом разряде 1.

Например:

```
unsigned char x=89,y=101;
unsigned char z=x^y;
//значение переменной z=60
```

Пояснение:

$x=89_{10}=01011001_2$
$y=101_{10}=01100101_2$
$x\text{^}y=00111100_2=60_{10}$

• << – сдвиг влево.

Так, a = b << c; осуществляется сдвиг значения *b* влево на *c* разрядов. В освободившиеся справа разряды *b* заносятся нули. Биты, «сдвинутые» за пределы разрядной сетки переменной, пропадают (рисунок 10.14).

Рисунок 10.14 – **Операция сдвига влево**

Сдвиг влево эквивалентен умножению на степень двойки, причем выполняется процессором гораздо быстрее.

Если операнд *c* отрицательный или больше длины операнда *b* в битах, то результат операции сдвига неопределен.

Операции сдвига не учитывают переполнение и потерю значимости.

Например:

```
signed char temp= 25<<2;
//значение переменной temp=100
```

Пояснение:

$25_{10}=00011001_2$
$25_{10}<<2=01100100_2=100_{10}=25*2^2$

```
signed char temp= -14 << 2;
//значение переменной temp=-56
```

Пояснение:

$-14_{10}=11110010_2$
$-14<<2=11001000_2=-56_{10}$

• >> – сдвиг вправо.

Так, a = b >> c; осуществляется сдвиг значения *b* вправо на *c* разрядов. Для заполнения позиций слева используется знаковый бит (ноль для положительных *b* и единица для отрицательных *b*). Биты, «сдвинутые» за пределы разрядной сетки переменной, пропадают (рисунок 10.15).

Сдвиг вправо положительных чисел эквивалентен делению на степень двойки, причем выполняется процессором гораздо быстрее.

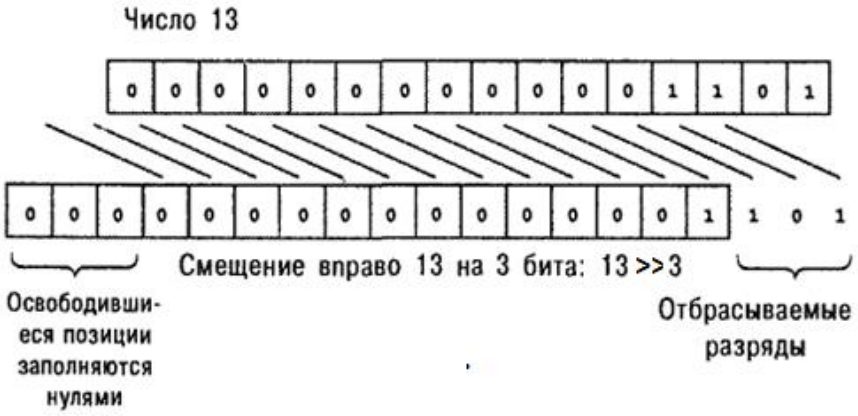


Рисунок 10.15 – **Операция сдвига вправо**

Например:

```
signed char temp= 25>>3;
//значение переменной temp=3
```

Пояснение:

$25_{10}=00011001_2$
$25_{10}>>3=00000011_2=3_{10}=25/2^3$

```
signed char temp= -14 >> 2;
//значение переменной temp=-4
```

Пояснение:

$-14_{10}=11110010_2$
$-14>>2=11111100_2=-4_{10}$

Рассмотрим несколько примеров использования битовых операций.

Пример 10.7. В программах управления аппаратурой часто бывает необходимо проверить, установлен ли определенный бит числа (т. е. равен ли он единице).

Для реализации этой цели создают маску – специальное число, имеющее единицы только в проверяемых разрядах, а остальные разряды – нулевые. Исследуемое число подвергают операции «битовое *и*» с маской. Если полученный результат равен 0, то бит не был установлен, а если отличен от 0, то был установлен.

В примере ниже проверяется второй бит справа (первый разряд). Напомним, что любое число, отличное от 0, преобразуется в булевское значение true:

```
#include <iostream>
using namespace std;
void main()
{    setlocale(LC_ALL,"rus");
     int mask=1<<1; //установлен второй бит справа
     int x;
     cout<<"Введите число: ";
     cin>>x;
     if(mask&x)
          cout<<"Второй бит установлен\n";
     else
          cout<<"Второй бит не установлен\n";
     system("pause");
}
```

Пример 10.8. Оригинальный способ проверить четное число или нет основан на том, что у нечетного числа в нулевом разряде бит равен 1, а у четного – 0. Поэтому в качестве маски используется 1:

```
#include <iostream>
using namespace std;
void main()
{    setlocale(LC_ALL,"rus");
     int x;
     cout<<"Введите число: ";
     cin>>x;
     if(x&1)
          cout<<"Нечетное\n";
     else
          cout<<"Четное\n";
     system("pause");
}
```

Пример 10.9. Операция ^ (битовое исключающее *или*) обладает одним важным свойством: если ее дважды применить к одному и тому же объекту, то она возвращает его в исходное состояние.

Это свойство часто используется для шифрования информации по ключу. Например, некто желает передать зашифрованное сообщение. Ко всем символам этого сообщения он применяет операцию ^ с фиксированным числом (ключом). В результате получается набор символов, который нельзя прочитать. Он передается получателю.

Получатель также знает ключ. Получив зашифрованную строку, он снова применяет к ней операцию ^ с ключом, и получает исходное послание.

```
#include <iostream>
#include <Windows.h>
using namespace std;
void main()
{
     SetConsoleCP(1251);
     SetConsoleOutputCP(1251);
     char key='Y';
     char str[20];
     cout<<"Введите строку не более 19 символов: ";
     gets(str);
     for(int i=0;i<strlen(str);i++)
          str[i]=str[i]^key;
     cout<<"Зашифрованная строка:";
     puts(str);
     for(int i=0;i<strlen(str);i++)
          str[i]=str[i]^key;
     cout<<"Восстановленная строка:";
     puts(str);
     system("pause");
}
```

Задачи

Задача 10.1. Описать структуру с именем Aeroflot, содержащую следующие поля:
- название пункта назначения рейса;
- номер рейса;
- тип самолета.

Написать программу, выполняющую следующие действия:

- ввод данных в массив, состоящий из четырех элементов типа Aeroflot;
- вывод списка рейсов, упорядоченных по возрастанию номера рейса;
- поиск по пункту назначения (пункт назначения запрашивается у пользователя). Выводится список рейсов, вылетающих в данный пункт.

Задача 10.2. Описать структуру Tovar (название товара, страна-производитель, цена). Создать массив товаров из 7 элементов. Инициализировать этот массив данными в программе.

Запросить у пользователя название страны и подсчитать среднюю цену на товары из этой страны. Вывести список товаров, упорядоченный по названию товара.

Задача 10.3. Разработать набор функций для работы с очередью (создание очереди (добавление одного первого элемента), добавление в конец очереди, извлечение из начала очереди, проверка пустоты очереди, печать содержимого очереди, подсчет количества элементов в очереди, очистка очереди). Проиллюстрировать их работу на примере.

Для самостоятельного решения

Задача 10.4. Описать структуру AVTO (номер, марка, объем двигателя, цвет автомобиля). Создать массив автомобилей из 5 элементов. Заполнить этот массив данными, запросив их у пользователя.

Запросить у пользователя цвет и распечатать все автомобили данного цвета.

Задача 10.5. Описать структуру Student (фамилия, возраст, средний балл). Создать массив студентов из 8 элементов. Массив инициализировать какими-либо значениями в программе.

Распечатать список студентов, упорядоченный по среднему баллу (отличники первые). Подсчитать количество студентов младше 20 лет.

Задача 10.6. Описать структуру VideoMag (название фильма, жанр, рейтинг популярности).

В программе создать массив, состоящий из восьми элементов типа VideoMag. Массив инициализировать какими-либо значениями в программе.

Написать программу, выполняющую следующие действия:

- поиск по жанру (название жанра запрашивается у пользователя);
- вывод информации о самом популярном фильме;
- вывод всех записей о фильмах на консоль, упорядоченных по рейтингу.

Задача 10.7. Описать структуру Student (фамилия, группа, успеваемость (массив из 5 int)). Создать массив студентов и написать программу, позволяющую:

- динамически изменять размер массива (добавлять или удалять запись о студенте с определенным номером);
- выводить список отличников (>75% отличных оценок);
- выводить список двоечников (>50% оценок 2 и 3);
- выводить список студентов, отсортированный по фамилии;
- выводить список студентов определенной группы, отсортированный по фамилии;
- выводить список студентов, отсортированный по среднему баллу (отличники первые);
- выполнять поиск по фамилии студента.

Задача 10.8. Описать структуру, содержащую битовые поля. Структура должна хранить информацию о конфигурации компьютера. Например, корпус AT – 0, ATX – 1; видео на борту – 0; видеокарта – 1 и т. д. Создать динамический массив записей о компьютерах и занести в него 5 записей, задав их значения в тексте программы. Реализовать следующие функции:

- печать всех записей;
- поиск и печать записей о компьютерах с корпусом ATX;
- добавления записи о компьютере;
- удаления записи о компьютере.

Задача 10.9. Разработать набор функций для работы с деком (создание дека, добавление в левый конец, добавление в правый конец, удаление с левого конца, удаление с правого конца, печать содержимого дека, очистка дека, подсчет количества элементов дека). Проиллюстрировать их работу на примере.

Задача 10.10. Разработать приложение «Фирма риелторов». Описать структуру «Квартира» (номер квартиры, количество комнат, общая площадь, стоимость, продана/свободна/забронирована). Создать массив из 10 квартир. Реализовать следующие функции:

- редактировать квартиру;
- продать квартиру;
- забронировать квартиру;
- печать всех квартир;
- печать всех проданных квартир;
- печать всех свободных квартир;
- печать всех забронированных квартир;
- расчет прибыли (сумма цены, всех проданных квартир).

Примечание – Рекомендуется задать исходную информацию о квартирах в программе.

ТЕМА 11. ФАЙЛЫ

11.1. Понятия файла и потока

Файл – именованная область памяти, в которой хранится информация. Обычно файл располагается на диске. В файлах хранятся данные, тексты программ, рисунки и т. д.

Физическая реализация файлов может быть различной и очень зависит от используемой операционной системы:

- некоторые системы хранят содержимое файла в одном месте, а информацию о нем – в другом месте (другие системы встраивают описание в сам файл);
- в текстовых файлах конец строки может отмечается двумя символами либо одним символом (например, в MS DOS конец строки отмечается символами '\r' и '\n' (возврат каретки и перевод строки), а в Macintosh – это один символ 'r');
- некоторые системы измеряют размер файла до ближайшего байта, а другие – блоками байтов.

Если учитываются эти и другие тонкости организации ввода и вывода, то это *низкоуровневый ввод и вывод,* который реализуется с помощью специальных библиотечных функций, ориентированных на конкретную операционную систему.

Если же используется *стандартный пакет ввода и вывода*, то все эти различия нивелируются самой системой, и программист работает с неким идеальным файлом. В нем текст состоит из последовательности строк, каждая строка заканчивается символом конца строки ('\n'). Если в MS-DOS на самом деле строки отделяются двумя символами '\r' и '\n', то система преобразует эти два символа в один при вводе и выполняет обратное преобразование при выводе. Поэтому для программиста конец строки выглядит как один символ.

Многие функции при достижении конца файла выдают символ EOF (End Of File). Символическая константа EOF определена в <stdio.h>. Обычно ее значение равно –1, но в выражениях следует писать EOF, чтобы не зависеть от конкретного значения.

Стандартный ввод и вывод соответствует стандарту ANSI и реализуется с помощью библиотечных функций, описанных в файле <stdio.h> (при работе с консолью это getchar(), gets() и scanf() для ввода, putchar(), puts(), printf() для вывода).

Поток – это абстракция, используемая для ввода и вывода данных в единой манере. Поток может быть связан с файлом на диске, с консолью, с принтером или сокетом для обмена данными по сети.

При работе с любой программой система автоматически открывает пять предопределенных потоков:

- stdin – поток для ввода, связанный с клавиатурой;

• stdout – поток для вывода на консоль, связанный с монитором (точнее, с окном консоли на мониторе);

• stderr – поток для вывода ошибок на консоль;

• stdaux – стандартный дополнительный поток;

• stdprn – стандартная печать.

Как будет показано ниже, стандартные потоки для ввода и вывода можно перенаправить. Например, можно сделать так, чтобы стандартный вывод шел не на консоль, а в файл. Или чтобы стандартный ввод забирал символы не с клавиатуры, а из файла на диске.

Но стандартный поток для вывода ошибок не может быть перенаправлен, ошибки всегда выводятся на экран монитора.

11.2. Текстовый и двоичный форматы файлов

Существует два формата файлов, установленных стандартом ANSI: текстовый и двоичный.

Файл *двоичного* формата – это последовательность байтов. Например, в таком файле хранятся числовые данные, которые предварительно не преобразуются в текст.

В файле *текстового* формата находятся символы, разделенные на строки. Файл текстового формата может быть открыт в текстовом редакторе, и вы увидите символы, которые можно прочесть. Если же в тестовом редакторе открыть файл двоичного формата, то вы увидите непонятный код.

На рисунке 11.1 показаны отличия в способах хранения числа 145 в двоичном и текстовом форматах.

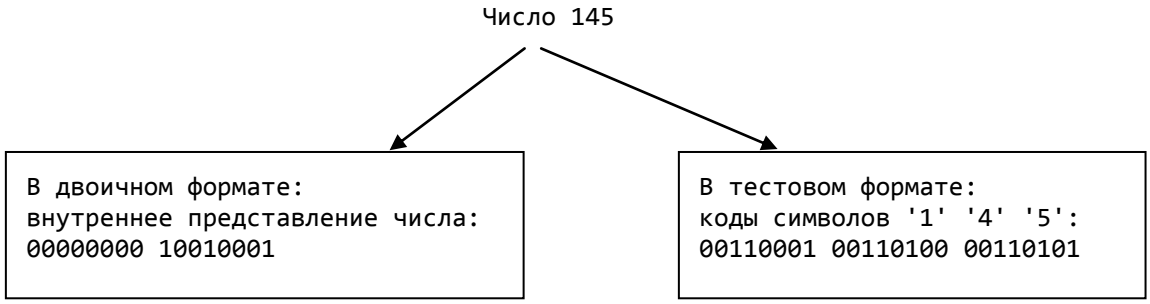


Рисунок 11.1 – **Представление данных в текстовом и двоичном формате**

Язык С имеет два режима открытия файлов: двоичный и текстовый. Обычно используемый режим соответствует формату файла. Однако можно открыть текстовый файл в двоичном режиме. Тогда можно будет прочесть коды всех хранящихся символов, без замены двух символов '\r' и '\n' на один. Открывать двоичный файл в текстовом режиме бессмысленно.

11.3. Буферизация ввода и вывода

Стандартный ввод и вывод использует буферизацию. Это означает, что данные из потока не сразу передаются в программу, а накапливаются в некоторой промежуточной памяти, называемой *буфером*. После заполнения буфера он передается программе.

Такой подход позволяет существенно повысить скорость работы программы. Размер буфера может быть 512, 1024 и более байтов.

Если выполняется ввод с клавиатуры, то буфер передается программе в тот момент, когда пользователь нажимает клавишу Enter. Кроме выигрыша в скорости, это еще дает возможность редактировать данные при вводе: пока не нажата клавиша Enter, возможно изменить содержимое строки.

Отличие буферизованного ввода от небуферизованного показано на рисунке 11.2. При выводе данных в файл или на консоль тоже может выполняться буферизация.

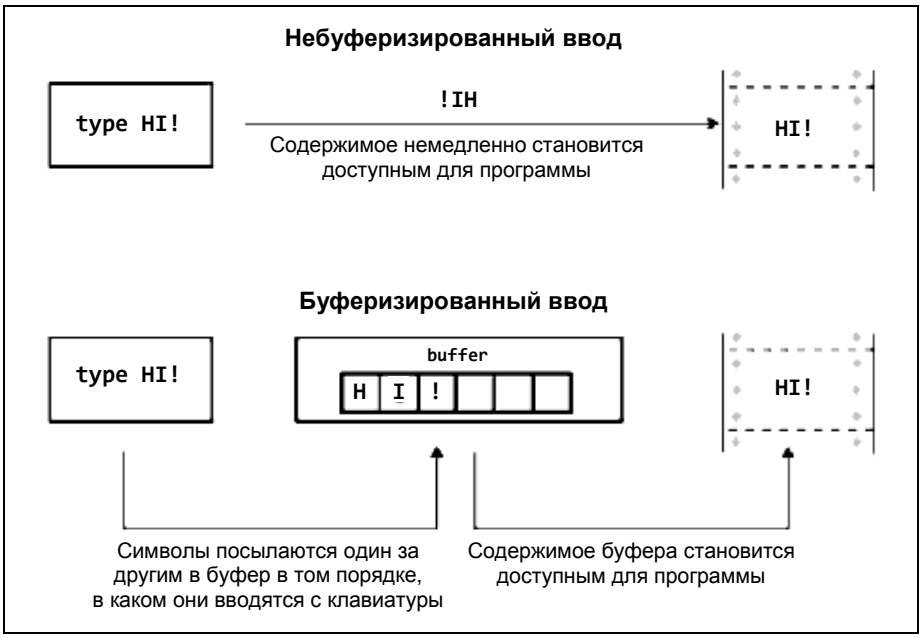


Рисунок 11.2 – **Буферизованный и небуферизованный ввод**

Функция `int getchar(void)` – это функция для ввода одного символа из стандартного потока ввода (по умолчанию – с клавиатуры).

Она возвращает следующий символ из потока или EOF в случае конца файла. Эта функция использует буферизацию ввода.

Функция `int putchar(int)` – функция для вывода одного символа в стандартный поток (по умолчанию – в окно консоли).

Она возвращает выведенный символ или EOF в случае ошибки. Обе данные функции описаны в заголовочном файле <stdio.h>.

Пример 11.1. Данный пример иллюстрирует эффект буферизации. Разработаем простую программу, которая повторяет вводимые символы:

```
#include <stdio.h>
void main()
{
     int c;
     while((c=getchar())!=EOF)
          putchar(c);
}
```

Символы считываются до тех пор, пока не будет обнаружен конец файла (при вводе с клавиатуры в MS DOS – это символ Ctrl+Z в начале строки).

Окно консоли для этой программы имеет вид, представленный на рисунке 11.3.

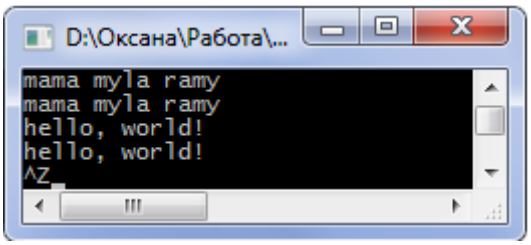


Рисунок 11.3 – **Окно консоли при буферизованном вводе**

Таким образом, хотя ввод и вывод в программе организованы посимвольно, на самом деле обработка символов начинается после того, как будет нажата клавиша Enter, и содержимое буфера станет доступно программе.

Обычно буферизованный ввод удобен или не заметен для пользователя. Однако существуют ситуации, когда при нажатии клавиши символ должен быть сразу же получен программой (например, в играх).

Для реализации *небуферизованного ввода* используются функции, объявленные в заголовочном файле <conio.h>:

```
int getche(void)
int getch(void)
```

Обе этих функции возвращают код введенного символа. При этом getche() повторяет нажатый на клавиатуре символ на экране консоли, а getch() – нет.

Часто функцию getch() используют для задержки экрана вместо system("pause"). Выполнение программы просто приостанавливается до тех пор, пока пользователь не нажмет какую-либо клавишу.

Эти функции не определены стандартом ANSI.

Пример 11.2. Небуферизованный ввод и вывод:

```
#include <conio.h>
#include <iostream>
using namespace std;
void main()
{
     char c;
     while((c=getche())!='#')
          putchar(c);
     getche();
}
```

Окно консоли в этом случае выглядит так, как показано на рисунке 11.4 (буквы повторяются сразу после ввода). Побочный эффект: не выдается символ конца файла, поэтому приходится использовать другой специальный символ.

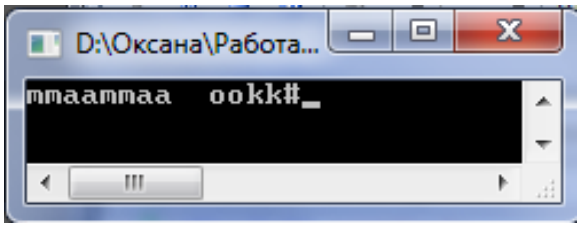


Рисунок 11.4 – **Окно консоли при небуферизованном вводе**

11.4. Перенаправление ввода и вывода

По умолчанию стандартный ввод получает данные от потока stdin, связанного с клавиатурой. Но стандартный поток можно перенаправить, т. е. связать stdin не с клавиатурой, а с каким-либо файлом на диске. Тогда функции getchar(), gets() и scanf() будут получать данные из этого файла. Так же себя ведет и потоковый ввод C++.

Аналогично можно перенаправить и стандартный поток вывода stdout, связав его с файлом на диске. Тогда функции putchar(), puts() и printf() (а также потоковый вывод cout) будут записывать данные в файл.

Поток для вывода ошибок и другие стандартные потоки перенаправить нельзя.

Перенаправить поток ввода можно с помощью символа < при вызове программы из командной строки. Следующая команда заставит приложение prog считывать символы из файла infile.txt, а не с клавиатуры:

```
prog.exe <infile.txt
```

Аналогично можно перенаправить поток вывода в файл вместо экрана консоли:

```
prog.exe >outfile.txt
```

Пример 11.3. Откомпилируем программу примера 1 и получим файл echo.exe (если открыть папку решения, то этот файл можно обнаружить в папке Debug). Поскольку мы собираемся запускать этот файл из командной строки, его имя не должно содержать пробелов (в командной строке действуют правила MS DOS, а не Windows).

Для удобства можем скопировать этот файл в другую папку (в данном случае была использована папка *Примеры*).

Запустим командную строку. Это можно сделать через кнопку *Пуск*, запустив программу cmd.exe. Другой способ открыть командную строку – в приложении Total Commander выбрать в меню *Команды* → *Открыть командную консоль*. Текущей директорией при этом будет та папка, которая в Total Commander была текущей.

Если в командной строке набрать имя файла, то программа будет выполнена так же, как если бы мы запускали ее из среды программирования (т. е. будет вводить данные с клавиатуры, а выводить в окно командной строки).

Если же набрать echo.exe>file2.txt (рисунок 11.5), то вывод будет перенаправлен в файл file2.txt (если он ранее не существовал, то он будет создан в этой директории). Все, что мы будем набирать на клавиатуре, будет записываться в файл.

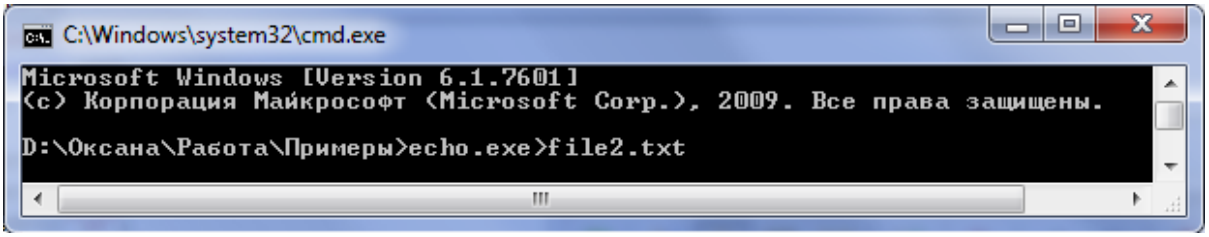


Рисунок 11.5 – **Перенаправление вывода**

Аналогично перенаправим поток ввода. Создадим заранее файл file1.txt, записав туда какой-либо текст. Тогда команда

```
echo.exe <file1.txt >file2.txt
```

даст возможность программе получать символы из файла file1.txt, а вывод направлять в файл file2.txt.

Порядок перечисления файлов при комбинированном перенаправлении не имеет значения. Возможны пробелы между именами файлов.

Перечислим некоторые правила перенаправления стандартных потоков:

- нельзя использовать один и тот же файл в качестве входного и выходного потоков (перед началом программы файл выходного потока удаляется и создается заново);
- операция перенаправления не может быть использована для соединения одного файла данных с другим (или одной программы с другой);
- входные данные можно брать только из одного файла, но не из нескольких (аналогично выходные данные могут быть направлены только в один файл);
- имя исполняемого файла не должно содержать пробелов.

Следует отметить, что в операционных системах Unix, Linux и MS DOS реализована также операция >>, которая позволяет добавлять данные в конец существующего файла.

11.5. Открытие файла

Перенаправление стандартных потоков ввода и вывода обладает существенным ограничением: нельзя часть информации вводить с клавиатуры, а часть – из файла. Аналогично нельзя разделить поток вывода, направив одни данные в файл, а другие на монитор.

Пусть, например, программа задает вопросы пользователю, а результаты записывает в файл. При перенаправлении потока в файл запишутся не только ответы, но и вопросы, а пользователь не увидит ничего.

Более удобным способом работы с файлом является выделение отдельных потоков для каждого вида данных. Прежде всего, такой поток следует открыть с помощью функции fopen() (объявлена в файле <stdio.h>). Эта функция берет внешнее имя, выполняет некоторые подготовительные операции, обменивается запросами с операционной системой (подробности сейчас не важны) и в итоге возвращает указатель, который в дальнейшем используется для чтения или записи. Ее формат:

```
FILE *fopen( const char *fname, const char *modeopen );
```

Строка `fname` – строка в стиле C, содержащая имя файла, который необходимо открыть. Этот параметр должен соответствовать правилам именования файлов в используемой системе, и может включать в себя путь, если система поддерживает его.

Параметр `modeopen` – строка, содержащая режим доступа к файлу.

В таблице 11.1 приведен список режимов и их описание.

Таблица 11.1 – **Режимы функции fopen()**

"r"	Открыть текстовый файл для чтения. Файл должен существовать
"w"	Открыть текстовый файл для записи. Если файл с таким именем уже существует, его содержимое стирается, и файл рассматривается как новый пустой файл. Если файла с таким именем нет, то он создается
"a"	Открыть текстовый файл для добавления данных в конец файла. Файл создается, если он не существует
"r+"	Открыть текстовый файл для обновления (чтения и записи). Этот файл должен существовать
"w+"	Открыть текстовый файл для обновления (записи и чтения). Если файл с таким именем уже существует, его содержимое стирается, и файл рассматривается как новый пустой файл. Если такого файла нет, то он создается
"a+"	Открыть текстовый файл для чтения и добавления данных. Все операции записи выполняются в конец файла, защищая предыдущее содержания файла от случайного изменения. Вы можете изменить позицию (FSEEK, перемотка назад) внутреннего указателя на любое место файла только для чтения, операции записи будет перемещать указатель в конец файла, и только после этого дописывать новую информацию. Файл создается, если он не существует
"rb", "wb", "ab", "r+b", "w+b", "a+b", "rb+", "wb+", "ab+"	Эти режимы подобны описанным выше, но вместо текстового режима доступа используется двоичный режим

Результат функции fopen() имеет тип «указатель на FILE», причем FILE – производный тип, объявленный в файле <stdio.h>. Фактически это структура, которая содержит информацию о файле: местонахождение буфера, текущую символьную позицию в буфере, характер операций (чтение или запись), наличие ошибок и состояние конца файла. Пользователю не обязательно знать эти детали, достаточно объявить переменную – указатель на файл, а затем присвоить ей результат, возвращаемый функцией fopen():

```
FILE *input;
input=fopen("myfile.txt","r")
```

Функция fopen() возвращает нулевой указатель, если она не может открыть файл. Неудача открытия файла может произойти по разным причинам: переполнение диска, неверное имя файла или путь к нему, ограничений доступа, сбой оборудования и т. д. В качественных программах ошибка открытия файла должна обрабатываться хотя бы выдачей сообщения:

```
if((input=fopen("myfile.txt","r"))==NULL)
{
     cout<<"Не удалось открыть файл для ввода\n";
```

```
        system("pause");
        exit(1);
    }
```

Еще лучше – проанализировать номер ошибки (errno объявлен в <errno.h>) или использовать специальные функции обработки ошибок, информацию о которых можно получить в справочной системе.

В Visual Studio 2013 и старше нужно использовать более современный вариант функции открытия файла:

```
fopen_s(&input, "myfile.txt", "r");
if (input == NULL) {
    cout << "Ошибка открытия файла ввода\n";
    system("pause");
    exit(1);
}
```

Два последних параметра этой функции имеют тот же смысл, что описано выше. Первый параметр – указатель на указатель на FILE. То есть в функцию передается адрес переменной, которая примет инициализированный этой функцией файловый указатель. Результат функции fopen_s – код ошибки при открытии файла (ноль в случае отсутствия ошибки).

11.6. Закрытие файла

Закрыть файл можно функцией `int fclose(FILE *fp)`.

Она разрывает связь между внешним именем и файловым указателем, установленную функцией fopen(), тем самым освобождая указатель для другого файла. При этом также очищается связанная с файлом буферная память, при необходимости хранящаяся там информация дописывается в файл или передается в программу.

Поскольку в большинстве операционных систем имеется ограничение на количество одновременно открытых программой файлов, полезно закрывать файлы и освобождать указатели, как только файлы становятся не нужны. Однако функция fclose автоматически вызывается для каждого открытого файла во время нормального завершения программы.

Функция fclose возвращает 0, если закрытие файла произошло успешно, и EOF в случае ошибки. Ошибка может произойти из-за того, что жесткий диск переполнен, диск или флешка изъяты из привода, произошла ошибка ввода и вывода, и т. д.

Пример закрытия файла и обработки соответствующей ошибки:

```
if(fclose(input)!=0)
{
    cout<<"ошибка закрытия файла ввода\n";
    system("pause");
    exit(1);
}
```

11.7. Функции чтения и записи в файл

Для посимвольного обмена данными с файлом используются функции *getc()*, которая возвращает очередной символ из файла, и *putc()*, которая выводит один символ в файл. Их формат:

```
int getc(FILE *fp)
int putc(int c, FILE *fp)
```

Параметр `fp` – это указатель на FILE, который должен быть предварительно инициализирован функцией fopen.

Функция getc() возвращает очередной символ из файла или EOF в случае ошибки или конца файла.

Функция putc() записывает символ *c* в файл *fp* и возвращает записанный символ или EOF в случае ошибки.

Обратите внимание, что считываемый и записываемый символ имеет тип int, а не char, т. е. размещается в четырех байтах. Это сделано потому, что кроме кода символа этот элемент должен хранить также код константы EOF, равный обычно –1. При записи и чтения собственно символа используется только младший байт этого значения типа int.

Пример использования функций getc() и putc():

```
FILE *fp, *fpout;
//открытие файлов
int ch;
ch=getc(fp); //считать символ из потока fp
putc(ch,fpout); //записать символ в поток fpout
```

Пример 11.4. Перепишем программу повторения символов из примера 11.1, используя функции работы с файлами. Программа посимвольно копирует данные из файла file1.txt в file2.txt. При этом будут выполняться проверки правильности открытия и закрытия каждого файла:

```
#include <iostream>
#include <stdio.h>
using namespace std;
void main()
{
     setlocale(LC_ALL,"rus");
     int c;
     FILE *input,*output;
//открытие и проверка корректности открытия файла file1.txt
     if((input=fopen("file1.txt","r"))==NULL)
     {
          cout<<"Не удалось открыть файл для ввода\n";
          system("pause");
          exit(1);
     }
//открытие и проверка корректности открытия файла file2.txt
     if((output=fopen("file2.txt","w"))==NULL)
     {
          cout<<"Не удалось открыть файл для вывода\n";
          system("pause");
          exit(1);
     }
//процесс посимвольного копирования
     cout<<"Копирую...\n
     c=getc(input); // ввод первого символа
     while(c!=EOF) //пока не конец файла
     {
          putc(c,output); //вывод символа
          c=getc(input); //ввод следующего
     }
//закрытие и проверка корректности закрытия файлов
     if(fclose(input)!=0)
     {
          cout<<"ошибка закрытия файла ввода\n";
          system("pause");
          exit(1);
     }
     if(fclose(output)!=0)
     {
          cout<<"ошибка закрытия файла вывода\n";
```

```
            system("pause");
            exit(1);
        }
        cout<<"Работа успешно завершена\n";
        system("pause");
}
```

Процесс посимвольного копирования можно описать более компактной конструкцией:

```
while((c=getc(input))!=EOF)
        putc(c,output);
```

При вычислении условия цикла while сначала вызывается функция getc() и результат ее работы (введенный символ) присваивается переменной *c*. Потом результат операции присваивания (т. е. считанный символ) сравнивается со значением EOF. Если символ не равен значению EOF (конец файла), то производится запись этого символа в поток output.

Для форматированного файлового ввода и вывода можно пользоваться функциями *fscanf()* и *fprintf()*. Они аналогичны функциям ввода scanf() и вывода printf(), и отличаются лишь наличием дополнительного первого параметра, который должен быть указателем на соответствующий файл:

```
int fscanf(FILE *fp, char *format, …)
int fprintf(FILE *fp, chat *format, …)
```

Функция fscanf() выдает целое значение, которое равно EOF при достижении конца файла.

Поэтому можно написать такой код копирования файла в задаче примера 11.4:

```
char str[128];
while(fscanf(input,"%s",str)!=EOF)
{
        fprintf(output,"%s ",str);
}
```

Функция fscanf() так же, как и функция scanf(), считывает строку до первого пробела или служебного символа ('\n', '\t' и т. д.).

Поэтому приведенный выше код считывает очередное слово из файла ввода и записывает его в файл вывода, добавляя только один пробел после слова. Таким образом, в преобразованном тексте исчезнут дополнительные пробелы между словами, знаки табуляции и конца строки.

Если такое преобразование недопустимо по условию задачи, нужно выполнять посимвольное копирование, либо воспользоваться функциями fgets() и fputs(), описанными ниже.

В Visual Studio 2013 (и более поздних версиях) следует использовать «безопасный» вариант функции форматированного ввода из файла, у которой имеется дополнительный параметр, ограничивающий размер вводимой строки:

```
while (fscanf_s(input,"%s",str,128)!= EOF) {
        fprintf(output,"%s",str);
}
```

Функции *fgets()* и *fputs()* аналогичны функциям gets() и puts(), которые работают со строкой, которая может включать пробелы и завершается символом перевода курсора.

Их формат:

```
char *fgets(char *str, int n, FILE *fp)
int fputs(const char *str, FILE *fp );
```

Здесь:

- `fp` – указатель на файл, с которым будет обмен данными;

• `str` – указатель на начало строки, предназначенной для размещения вводимых данных (или указатель на выводимую строку);

• `n` – целое число, определяющее максимальный размер входной строки.

Функция fgets() читает не более *n* – 1 символов из файла. Завершает работу, если прочитано заданное количество символов, либо достигнут конец строки или конец файла. В конец введенного массива символов записывается нуль-терминатор ('\0').

Функция fgets() не удаляет символ конца строки при чтении, а помещает его в массив символов. Этим она отличается от функции gets(), которая считывает символ конца строки, но отбрасывает его за ненадобностью. Это отличие проиллюстрировано на рисунке 11.6.

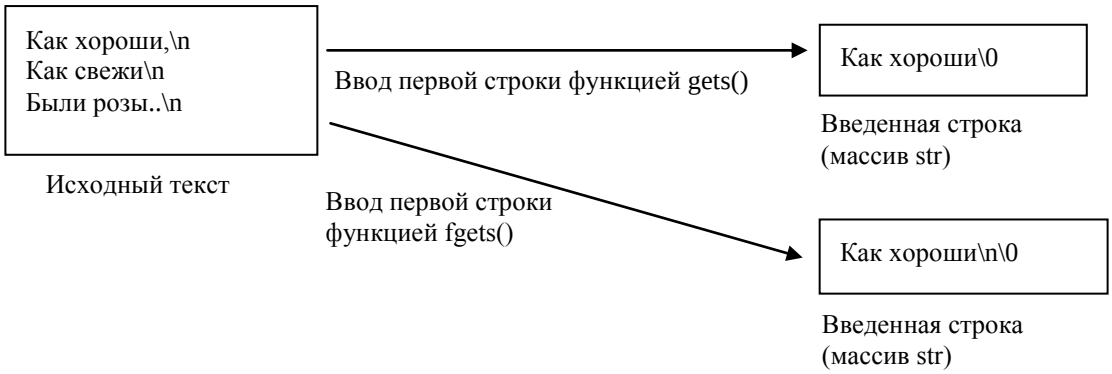


Рисунок 11.6 – **Обработка символа конца строки функциями gets() и fgets()**

Функция fgets() возвращает значение NULL, если происходит ошибка, либо если достигнут конец файла, и при этом ни один символ не был считан. В случае удачного ввода строки она возвращает ее адрес в памяти.

Функция fputs() (в отличие от функции puts) не добавляет символ конца строки при выводе, а выводит содержимое строки так, как оно есть. Поскольку функция fgets() сохраняет символ конца строки, а функция fputs() его не добавляет, они довольно хорошо работают в паре.

В случае успешного вывода функция fputs() возвращает неотрицательное значение. В случае ошибки она возвращает значение EOF.

Перепишем процесс копирования из примера 11.4 с помощью функций fgets() и fputs(). Результирующий файл будет точной копией исходного (включая все пробелы, знаки табуляции и начало новой строки):

```
const int MAX=128;
char str[MAX];
while(fgets(str, MAX, input)!=NULL)
{
     fputs(str,output);
}
```

Пример 11.5. Данные о студентах (фамилия, имя, группа, средний балл) записаны в текстовом файле data.txt. Каждая запись находится в отдельной строке, поля разделены пробелами. Первая строка содержит количество записей. Пример содержимого файла:

3

Иванов Сергей G11 8.5

Кириллов Андрей G11 6.5

Алексеева Ольга G21 8.9

Необходимо прочитать данные в массив структур, вывести их на консоль. Затем создать новый файл с именем g11.txt и переписать в него записи о всех студентах, которые учатся в группе G11. В первой строке должно быть записано количество таких студентов.

Для тестирования программы нужно создать файл data.txt в текстовом редакторе.

Текст программы приведен ниже. Для удобства отладки ввод записи о студенте, вывод массива студентов и запись данных о студенте в файл оформлены отдельными функциями.

Парсинг строки аналогичен примеру использования функции strtok, приведенном в теме 9. Программа работает в Visual Studio 2015, поэтому в ней использованы современные «безопасные» варианты функций работы со строками:

```
#include <iostream>
using namespace std;
const int MAX = 50;
struct Student {
	char fam[MAX];
	char name[MAX];
	char gruppa[MAX];
	double ball;
};
int takeStudent(FILE *input, Student *person);
void printSpisok(Student *spisok, int n);
void putStudent(FILE *output, Student person);
void main() {
	setlocale(LC_ALL, "rus");
	Student* spisok; //указатель на начало массива студентов
	int n;//количество студентов в массиве
	FILE *input = NULL, *output = NULL; //файловые указатели
	int kolG11 = 0;//счетчик студентов из группы G11
	//открытие исходного файла
	fopen_s(&input, "data.txt", "r");
	if (input == NULL) {
		cout << "Не удается открыть файл ввода\n";
		system("pause");
		exit(0);
	}
	fscanf_s(input, "%d\n", &n);//считать из файла количество записей
	spisok = new Student[n]; //выделить память под массив студентов
	for (int i = 0; i < n; i++) {
		takeStudent(input, spisok + i);
		//подсчет студентов из группы G11
		if (strcmp(spisok[i].gruppa, "G11") == 0)
			kolG11++;
	}
	printSpisok(spisok, n);
	fopen_s(&output, "g11.txt", "w");
	if (output == NULL) {
		cout << "Не удается открыть файл вывода\n";
		system("pause");
		exit(0);
	}
	fprintf(output, "%d\n", kolG11);//запись кол-ва студентов в файл
	//запись данных о студентах G11
	for (int i = 0; i < n; i++) {
		if (strcmp(spisok[i].gruppa, "G11") == 0) {
			putStudent(output, spisok[i]);//запись в файл
		}
	}
	//закрытие файлов
	fclose(input);
	fclose(output);
	system("pause");
}
//ввод строки и парсинг (разбор строки на элементы записи о студенте)
//результат функции -1, если достигнут конец файла
// и 0, если ввод удачен
int takeStudent(FILE *input, Student *person) {
	char str[4 * MAX]; //строка-буфер для чтения одной записи
	char *word,*next; //вспомогательные указатели при разборе строки
	//чтение строки из файла
```

```
     if(fgets(str, 4*MAX, input)==NULL)
          return -1; //возврат, если конец файла
     //разбор строки
     word=strtok_s(str, " ", &next);
     strcpy_s(person->fam, word);
     word = strtok_s(NULL, " ", &next);
     strcpy_s(person->name, word);
     word = strtok_s(NULL, " ", &next);
     strcpy_s(person->gruppa, word);
     word = strtok_s(NULL, " ", &next);
     person->ball = atof(word);
     return 0;
}
//печать массива студентов на консоль
void printSpisok(Student *spisok, int n) {
     for (int i = 0; i < n; i++) {
          cout << spisok[i].fam << " " << spisok[i].name<<" ";
          cout << spisok[i].gruppa << " " << spisok[i].ball << "\n";
     }
}
//запись информации о студенте в файл
void putStudent(FILE *output, Student person) {
     fprintf(output, "%s %s ", person.fam, person.name);
     fprintf(output, "%s %3.1f\n", person.gruppa, person.ball);
}
```

11.8. Произвольный доступ к файлу

Рассмотренные функции файлового ввода и вывода позволяют реализовать последовательный доступ к файлу. Существует возможность получить произвольный доступ к файлу, т. е. обращаться к какому-то месту в середине файла. С этой целью используются функции fseek() и ftell().

Функция *fseek()* рассматривает файл как массив. Понятие текущей позиции в файле аналогично понятию индекса. Функция fseek() устанавливает текущую позицию, которая определяется путем добавления смещения к исходному положению. Формат функции:

```
int fseek( FILE *filestream, long int offset, int origin )
```

Параметры:

- `filestream` – указатель на FILE, идентифицирующий поток;
- `offset` – количество байт для смещения относительно некоторого положения, заданного в третьем параметре (этот параметр должен иметь тип long);
- `origin` – позиция в файле, относительно которой будет выполняться смещение. Данная позиция задается одной из следующих констант, определенных в файле <cstdio>:

```
SEEK_SET      Начало файла
SEEK_CUR      Текущее положение позиции в файле
SEEK_END      Конец файла
```

В случае успешного выполнения позиционирования функция fseek() возвращает нулевое значение, в случае неудачи – ненулевое значение.

Рассмотрим несколько примеров использования функции fseek() (fp – некоторый указатель на FILE):

```
fseek(fp, 0L, SEEK_SET); //перейти в начало файла
fseek(fp, 10L, SEEK_SET); //сместиться на 10 байт относ. начала файла
fseek(fp, 2L, SEEK_CUR); //сместиться на 2 байта относ.текущей позиции
```

```
fseek(fp, 0L, SEEK_END); //перейти в конец файла
fseek(fp, -10L, SEEK_END); //сместиться на 10 байт от конца файла к
                           //началу
```

Функция *ftell()* возвращает значение текущей позиции в файле. Формат функции:

```
long int ftell( FILE *filestream ).
```

`filestream` – указатель на FILE, идентифицирующий поток.

В случае успеха возвращается текущее значение индикатора положения. Если происходит ошибка, то возвращается значение –1L.

Первоначально в ОС Unix функция ftell() возвращала позицию символа в файле, причем первый байт получал номер позиции 0. В ANSI C это определение применимо в файлам, открытым в двоичном режиме. В текстовом режиме результат функции ftell() – не обязательно количество символов от начала файла. Поэтому с этой функцией лучше всего работать при использовании двоичного представления файла.

Тем не менее стандарт ANSI утверждает, что и в текстовом режиме функция ftell() возвращает значение, которое может быть использовано как второй аргумент функции fseek().

Отметим, что при чтении или записи в файл одного байта автоматически увеличивается номер текущей позиции.

Пример 11.6. Определение размера файла с помощью функций fseek() и ftell(). Программа выводит размер файла на консоль:

```
#include <iostream>
using namespace std;
void main()
{
     setlocale(LC_ALL,"rus");
     FILE *fp;
     if((fp=fopen("mytext.txt","rb"))==NULL) //файл для чтения в
                                             // двоичном режиме
     {
          cout<<"Не удается открыть файл mytext.txt\n";
          system("pause");
          exit(1);
     }
     fseek(fp,0L,SEEK_END); //текущая позиция в конец файла
     long k=ftell(fp); //вернуть текущее положение
     cout<<"Размер файла= "<<k<<" байт\n";
     system("pause");
}
```

Пример 11.7. Программа выводит на консоль файл «наоборот»:

```
#include <iostream>
#include <stdio.h>
using namespace std;
void main()
{
     setlocale(LC_ALL,"rus");
     char ch;
     FILE *input;
     if((input=fopen("file1.txt","rb"))==NULL) //файл для чтения в
                                               //двоичном режиме
     {
          cout<<"Не удалось открыть файл для ввода\n";
          system("pause");
          exit(1);
     }
     fseek(input,0L,SEEK_END); //позиционирование в конец файла;
```

```
     int last=ftell(input); //номер последнего байта
                           //теперь знаем количество байт
     for(long int i=1L;i<=last;i++)
     {
          fseek(input, -i,SEEK_END);//вернуться на i назад от конца
          ch=fgetc(input); //чтение символа из тек. позиции
          if(ch!='\r') putchar(ch);//В DOS пропускаем первый
                                   //символ новой строки
     }
     putchar('\n');
     system("pause");
}
```

11.9. Использование аргументов командной строки

При запуске программы из командной строки ей можно передавать аргументы, указывая их через пробел. Например, можно в качестве аргумента указать имя файла, предназначенного для вывода результата работы программы:

```
echo.exe file3.txt
```

Чтобы в программе обработать аргументы командной строки, необходимо указать параметры функции main:

```
void main(int argc, char* argv[])
```

Первый параметр argc содержит количество параметров, передаваемых в функцию main(). Причем argc всегда не меньше 1, даже когда мы не передаем никакой информации, поскольку первым параметром считается имя программы.

Параметр argv[] – массив указателей на строки, соответствующие параметрам. То есть в примере вызова, который приведен выше, argv[0] – это имя программы «echo.exe», а argv[1] – это имя файла «file3.txt», который мы предназначили для вывода.

Пример 11.8. Программа открывает для записи файл, который указан как аргумент командной строки. Затем вводит строку с клавиатуры и записывает ее в файл:

```
#define _CRT_SECURE_NO_WARNINGS
#include <conio.h>
#include <iostream>
using namespace std;
#define MAXLEN 128
void main(int argc, char* argv[])
{
     setlocale(LC_ALL,"rus");
     char str[MAXLEN];
     //проверка наличия аргументов командной строки
     if(argc==1)
     {
          printf("Нет имени файла при запуске %s \n",argv[0]);
          _getch();
          exit(1);
     }
     FILE *fp;
//открываем файл, имя которого является параметром командной строки
     if((fp=fopen(argv[1],"w"))==NULL)
     {
          printf("Файл вывода %s не может быть открыт\n",argv[1]);
          _getch();
          exit(1);
```

```
        }
        printf("Введите строку: \n");
        gets(str);
        //запись в файл
        fprintf(fp,str);
}
```

В программе, как правило, сначала проверяется наличие дополнительных аргументов командной строки. В примере выше в том случае, если программа запущена без аргументов (например, из среды Visual Studio), она выводит об этом сообщение и прекращает работу.

Если же имеется дополнительный аргумент командной строки, то он используется как имя файла в функции fopen(). Далее в демонстрационных целях в этот файл записывается строка, которую пользователь вводит с клавиатуры.

Чтобы приведенная ниже программа компилировалась и в Visual Studio 2010, и в Visual Studio 2013, в ней использован «безопасный» вариант функции _getch().

Кроме того, объявлена константа `_CRT_SECURE_NO_WARNINGS`, которая сообщает компилятору о необходимости пропускать «устаревшие» с точки зрения Visual Studio 2013 варианты функций работы с файлами (fopen(), fprintf() и т. д.). Соответствующая директива #define должна стоять самой первой, до всех #include.

Пример запуска этой программы из командной строки показан на рисунке 11.7.

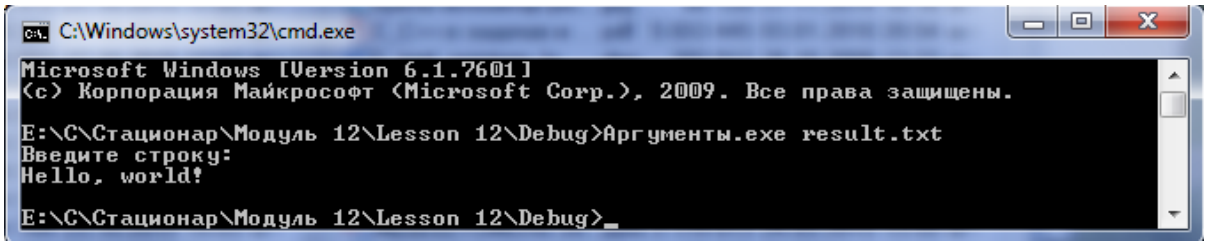


Рисунок 11.7 – **Использование аргументов командной строки**

11.10. Использование файлов в двоичном формате

В двоичных файлах хранится последовательность байт. Таким образом, можно поместить в двоичный файл данные в том же виде, в каком они хранятся в памяти компьютера (не преобразуя их в символьное представление, как в случае текстового файла). Разницу между двоичным и текстовым форматом иллюстрирует рисунок 11.8.

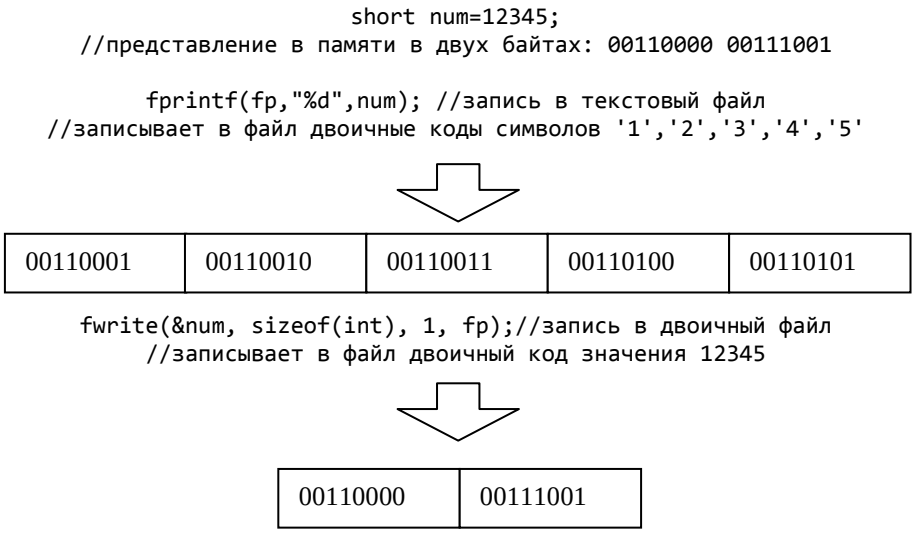


Рисунок 11.8 – **Отличие текстового и двоичного формата при записи чисел в файл**

Для работы с файлом в двоичном формате при открытии файла вторым параметром функции fopen() нужно указывать режим с буквой «b»: например, для записи "wb", для чтения "rb".

Для чтения и записи в двоичный файл используются специальные функции fread() и fwrite().

Функция *fwrite()* записывает двоичные данные в файл. Имеет следующий формат:

```
size_t fwrite(const void *ptrvoid, size_t size, size_t count, FILE
*filestream );
```

Функция возвращает количество элементов, которые были записаны в файл. При успешном выполнении функции возвращаемое значение должно совпадать с третьим параметром.

Параметры функции:

- `ptrvoid` – указатель на массив элементов, которые необходимо записать в файл;
- `size` – размер в байтах каждого элемента массива;
- `count` – количество элементов, каждый из которых занимает size байт;
- `filestream` – указатель на FILE, который связан с потоком вывода.

После выполнения функции текущая позиция в файле увеличивается на число записанных байт.

Обратите внимание, что тип первого параметра в прототипе функции – void. Этот тип подходит для любого другого типа языка С. Таким образом, первый параметр функции может иметь любой тип (как стандартный, так и производный). Например:

```
//в файл записывается массив символов
char buffer[] = { 'w' , 't' , 'f' };
fwrite(buffer , 1 , sizeof(buffer) , fptr);

//в файл записывается массив целых чисел
int temp[] = {0,1,2,3,4,5,6,7,8,9};
fwrite(temp, sizeof(temp), 1,fp);

//в файл записывается одно вещественное число
double value=3.5;
fwrite(&value,sizeof(double),1,output);
```

Функция *fread()* считывает двоичные данные из файла и имеет следующий формат:

```
size_t fread(void *ptrvoid, size_t size, size_t count, FILE *filestream );
```

Считывается массив размером `count` элементов, каждый из которых имеет размер `size` байт, из потока `filestream`, и сохраняет его в блоке памяти, на который указывает `ptrvoid`.

Текущая позиция в файле увеличивается на общее число записанных байтов.

Функция возвращает общее число считанных элементов. Возвращаемое значение должно совпадать с count. Если это не так, то произошла ошибка либо был достигнут конец файла. Какое именно из этих событий произошло, можно определить с помощью функций ferror() или feof().

Функция `int ferror(FILE * filestream)` возвращает значение, отличное от нуля, если при выполнении предыдущей функции ввода и вывода произошла ошибка.

Функция `int feof (FILE * filestream)` возвращает значение, отличное от нуля, если конец файла был достигнут при выполнении предыдущей операции ввода и вывода.

Например:

```
// считываем массив символов
char buffer[lSize];
size_t result = fread(buffer, 1, lSize, ptrFile);
if (result != lSize)
{
     fputs("Ошибка чтения", stderr);
```

```
     exit (3);
}

//считываем массив целых чисел
int numbers[10];
fread(numbers, sizeof(int), 10, fp);

//считываем одно вещественное число
double data;
fread(&data,sizeof(double),1,input);
```

Пример 11.9. Программа демонстрирует работу с двоичным файлом. У пользователя запрашивается имя файла, в который записываются десять случайных целых чисел.

Затем производится чтение из этого файла всех записанных данных от начала до конца.

И в завершении демонстрируется произвольный доступ к файлу: у пользователя запрашивается индекс одного элемента в файле, указатель файла устанавливается на соответствующее место, нужное число считывается и выводится на экран.

```
#include <stdio.h>
#include <iostream>
#include <conio.h>
#include <time.h>
using namespace std;
#define SIZE 10
int main()
{
     setlocale(LC_ALL,"rus");
//заполнение массива случайными числами
     int temp[SIZE];
     srand(time(0));
     for(int i=0;i<SIZE;i++)
          temp[i]=rand()%100+1;
     FILE *output, *input;
     char filename[128];
     puts("Введите имя файла: ");
     gets(filename);
     //попытка открыть файл для записи
     if((output=fopen(filename,"wb"))==NULL)
     {
          puts("Невозможно открыть для записи файл");
          puts(filename);
          _getch();
          return(1);
     }
     //запись данных в файл (записывается сразу весь массив)
     fwrite(temp, sizeof(temp), 1, output);
     fclose(output);
     //Открываем файл в режиме бинарного чтения:
     input = fopen(filename, "rb");
     if (!input) cout << "Невозможно открыть файл для чтения!\n";
     else
     {
          /* Определяем размер файла в БАЙТАХ
          fseek(input, 0, SEEK_END);
          int fsize = ftell(input);
          // Определяем количество int в файле
          int num = fsize / sizeof(int);
          //выделяем память под целочисленный массив для данных из файла
          int *numbers = new int[num];
          //Считываем данные в массив (сразу все)
          fseek(input, 0, SEEK_SET);
          fread(numbers, sizeof(int), num, input);
```

```
            // Выводим содержимое массива
            for (int i = 0; i < num; ++i)
                 cout << numbers[i] << endl;
            //произвольный доступ к массиву:
            int k=0, value;
            printf("Введите индекс элемента: ");
            scanf("%d",&k);
            //рассчитываем позицию первого байта числа
            long poz=(long)k*sizeof(int);
            //установка текущей позиции в файле
            fseek(input,poz,SEEK_SET);
            // чтение одного числа
            fread(&value,sizeof(int),1,input);
            printf("Элемент равен= %d \n",value);
            fclose(input);
      }
      _getch();
}
```

Задачи

Задача 11.1. Программа подсчитывает количество символов в текстовом файле. Реализуйте программу двумя способами: а) программа запускается из командной строки с использованием перенаправления потока ввода; б) программа запрашивает имя анализируемого файла.

Задача 11.2. Программа удаляет из тестового файла все слова «help!» и подсчитывает количество оставшихся слов. Имя исходного файла и имя файла-результата запрашиваются у пользователя. Количество слов выводится на консоль. Все лишние пробелы между словами могут быть удалены.

Задача 11.3. Данные о товарах (название товара, страна-производитель, цена) записаны в текстовом файле. Каждая запись в отдельной строке, поля разделены пробелами. Первая строка содержит количество записей. Пример содержимого файла:

3
Milk Belarus 10.900
Shugar Belarus12.300
Kola Russia 15.400

Для тестирования программы создайте подобный файл с данными в текстовом редакторе.

Разработайте приложение, которое при запуске запрашивает имя файла с данными, создает массив структур в динамической памяти и считывает в него данные из файла. Затем выводится меню, которое предлагает пользователю выбрать вариант из следующих действий:

- распечатать список товаров без сортировки;
- распечатать список товаров, упорядоченный по неубыванию цены;
- добавить товар;
- удалить товар;
- распечатать список товаров из некоторой страны (страна-производитель затем запрашивается у пользователя);
- сохранить данные в файл (имя файла для записи запрашивается у пользователя);
- завершение работы (перед выходом из программы пользователю предлагается сохранить обновленные данные в файле с тем же именем, которое использовалось для загрузки данных в начале программы, либо выйти без сохранения изменений).

Код реализации каждого пункта меню оформить в виде отдельной функции.

Для самостоятельного решения

Задача 11.4. Реализовать задачу из примера 11.3, но данные хранить не в текстовом, а в двоичном файле (в файл записывается структура целиком).

Меню должно иметь следующие пункты:

- ввод данных о товарах с консоли;
- сохранение данных в файл (если данные еще не были введены, то должно выдаваться сообщение);
- чтение данных из файла (если массив структур уже заполнен, то должно выдаваться предупреждение «Старые данные удалены»);
- сортировка массива товаров по наименованию;
- выход из программы.

Задача 11.5. Написать две функции, одна из которых вызывается для шифрования текста шифром Цезаря (имя исходного файла и имя файла-результата, а также ключ шифрования передаются как параметры функции). Вторая функция используется для расшифровки (набор параметров аналогичный). Функцию main использовать для демонстрации возможностей набора функций шифрования.

Шифр Цезаря – один из древнейших шифров. При шифровании каждый символ заменяется другим, код которого больше на значение ключа.

Например, шифрование с использованием ключа 3:

Оригинальный текст: `Съешь же еще этих мягких французских булок да выпей чаю.`

Шифрованный текст: `Фэзыя йз зьи ахлш пвенлш чугрщцкфнлш дцосн жг еютзм ъгб.`

Задача 11.6. Дан текстовый файл (имя файла запрашивается у пользователя). На место первой строки с пробелами записать строку из двадцати черточек («----------»). Если строк с пробелами нет, то записать черточки в конце файла. Должен измениться исходный файл.

СОДЕРЖАНИЕ

Учебное издание

ОСНОВЫ АЛГОРИТМИЗАЦИИ И ПРОГРАММИРОВАНИЯ

Пособие
для реализации содержания образовательных программ
высшего образования I ступени

В двух частях

Часть 2

Автор-составитель
Еськова Оксана Ивановна

Редактор Т. В. Гавриленко
Компьютерная верстка Л. Ф. Барановская

Подписано в печать 19.03.18. Формат 60 × 84 $^{1}/_{8}$.
Бумага офсетная. Гарнитура Таймс. Ризография.
Усл. печ. л. 12,09. Уч.-изд. л. 10,58. Тираж 42 экз.
Заказ №

Издатель и полиграфическое исполнение:
учреждение образования «Белорусский торгово-экономический
университет потребительской кооперации».
Свидетельство о государственной регистрации издателя, изготовителя,
распространителя печатных изданий
№ 1/138 от 08.01.2014.
Просп. Октября, 50, 246029, Гомель.
http://www.i-bteu.by

БЕЛКООПСОЮЗ
УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ
«БЕЛОРУССКИЙ ТОРГОВО-ЭКОНОМИЧЕСКИЙ
УНИВЕРСИТЕТ ПОТРЕБИТЕЛЬСКОЙ КООПЕРАЦИИ»

Кафедра информационно-вычислительных систем

ОСНОВЫ АЛГОРИТМИЗАЦИИ И ПРОГРАММИРОВАНИЯ

Пособие
для реализации содержания образовательных программ
высшего образования I ступени

В двух частях

Часть 2

Гомель 2018